

RP2040 Datasheet

A microcontroller
by Raspberry Pi

版权页

© 2020-2025 Raspberry Pi Ltd (前称 Raspberry Pi (Trading) Ltd.)

本文件依据知识共享署名-禁止演绎 4.0 国际许可协议 (CC BY-ND) 授权。

部分版权归 Synopsys, Inc. © 2019 所有

版权所有。经许可使用。Synopsys 与 DesignWare 均为 Synopsys, Inc. 注册商标。部分版权归 Arm Limited © 2000-2001、2005、2007、2009、2011-2012、2016 所有。

版权所有。经许可使用。

构建日期: 2025-02-2

0 构建版本: 3184e62-clean

法律免责声明通知

Raspberry Pi 产品（包括数据表）不时修订的技术和可靠性数据（以下简称“资源”）由 Raspberry Pi Ltd（以下简称“RPL”）按“现状”提供，并否认任何明示或暗示的保证，包括但不限于对适销性及特定用途适用性的暗示保证。在适用法律允许的最大范围内，无论基于合同、严格责任或侵权（包括过失或其他）理论，RPL 对因使用该资源所引起的任何直接、间接、附带、特殊、惩罚性或后续性损害（包括但不限于采购替代商品或服务；使用权丧失、数据或利润损失；或业务中断）均不承担任何责任，即便已被告知发生此类损害的可能性。

RPL保留随时对资源（RESOURCES）或其中所描述的任何产品进行任何增强、改进、更正或其他修改的权利，且无需另行通知。

该资源（RESOURCES）仅供具备相应设计知识水平的熟练用户使用。用户应自行承担对资源（RESOURCES）的选择和使用以及对其中所描述产品的任何应用所产生的全部责任。用户同意赔偿并使RPL免受因其使用资源（RESOURCES）而产生的所有责任、费用、损害或其他损失的影响。

RPL授权用户仅可将资源（RESOURCES）与Raspberry Pi产品结合使用。禁止对资源（RESOURCES）作任何其他用途。未授予RPL或任何第三方的其他知识产权许可。

高风险活动。Raspberry Pi产品并非为在要求故障安全性能的危险环境中使用而设计、制造或意图使用，如核设施运行、航空导航或通信系统、空中交通管制、武器系统或安全关键应用（包括生命维持系统及其他医疗设备），在这些环境中产品的故障可能直接导致死亡、人身伤害或严重的身体或环境损害（以下简称“高风险活动”）。RPL明确否认对用于高风险活动的适用性的任何明示或暗示保证，且对将Raspberry Pi产品用于高风险活动或包含于其中不承担任何责任。

Raspberry Pi 产品依据 RPL 标准条款提供。RPL 提供的资源（RESOURCES）不构成对 RPL 标准条款的扩展或修改，包括但不限于其中明确表达的免责声明及保证条款。

目录

版权页	1
法律免责声明通知	1
1. 引言	8
1.1. 芯片为何命名为 RP2040?	8
1.2. 概要	9
1.3. 芯片	9
1.4. 引脚参考	10
1.4.1. 引脚位置	10
1.4.2. 引脚描述	11
1.4.3. GPIO 功能	12
2. 系统描述	14
2.1. 总线结构	14
2.1.1. AHB-Lite 交叉开关	15
2.1.2. 原子寄存器访问	17
2.1.3. APB 桥接	17
2.1.4. 窄带 IO 寄存器写入	17
2.1.5. 寄存器列表	18
2.2. 地址映射	24
2.2.1. 概述	24
2.2.2. 详细说明	24
2.3. 处理器子系统	26
2.3.1. SIO	27
2.3.2. 中断	60
2.3.3. 事件信号	61
2.3.4. 调试	61
2.4. Cortex-M0+	62
2.4.1. 功能特性	62
2.4.2. 功能描述	64
2.4.3. 程序员模型	68
2.4.4. 系统控制	73
2.4.5. NVIC	74
2.4.6. MPU	76
2.4.7. 调试	76
2.4.8. 寄存器列表	77
2.5. DMA	91
2.5.1. 通道配置	91
2.5.2. 启动通道	93
2.5.3. 数据请求 (DREQ)	95
2.5.4. 中断	96
2.5.5. 附加功能	96
2.5.6. 示例用例	97
2.5.7. 寄存器列表	101
2.6. 内存	120
2.6.1. 只读存储器 (ROM)	120
2.6.2. 静态随机存取存储器 (SRAM)	121
2.6.3. 闪存	122
2.7. 启动顺序	128
2.8. 启动只读存储器 (Bootrom)	128
2.8.1. 处理器控制的启动顺序	129
2.8.2. 在处理器核1上启动代码	131
2.8.3. 启动只读存储器内容	132
2.8.4. USB大容量存储设备接口	143
2.8.5. USB PICOBLOCK 接口	144
2.9. 电源供应	150
2.9.1. 数字IO供电 (IOVDD)	151

2.9.2. 数字核心供电 (DVDD)	151
2.9.3. 片上电压调节器输入电源 (VREG_VIN)	151
2.9.4. USB PHY 供电 (USB_VDD)	151
2.9.5. ADC 供电 (ADC_AVDD)	152
2.9.6. 电源顺序	152
2.9.7. 电源方案	152
2.10. 核心供电调节器	155
2.10.1. 应用电路	155
2.10.2. 工作模式	156
2.10.3. 输出电压选择	157
2.10.4. 状态	157
2.10.5. 电流限制	157
2.10.6. 寄存器列表	157
2.10.7. 详细规格	160
2.11. 电源控制	160
2.11.1. 顶层时钟门控	160
2.11.2. SLEEP 状态	161
2.11.3. 休眠状态	161
2.11.4. 内存断电	161
2.11.5. 程序员模型	162
2.12. 芯片级复位	163
2.12.1. 概述	163
2.12.2. 上电复位	164
2.12.3. 欠压检测	165
2.12.4. 电源监控	167
2.12.5. 外部复位	167
2.12.6. 救援调试端口复位	167
2.12.7. 最近一次复位来源	168
2.12.8. 寄存器列表	168
2.13. 上电状态机	168
2.13.1. 概述	168
2.13.2. 上电序列	168
2.13.3. 寄存器控制	169
2.13.4. 与看门狗的交互	169
2.13.5. 寄存器列表	169
2.14. 子系统复位	172
2.14.1. 概述	172
2.14.2. 程序员模型	173
2.14.3. 寄存器列表	175
2.15. 时钟	178
2.15.1. 概述	178
2.15.2. 时钟源	179
2.15.3. 时钟发生器	183
2.15.4. 频率计数器	186
2.15.5. Resus	187
2.15.6. 程序员模型	187
2.15.7. 寄存器列表	194
2.16. 晶体振荡器 (XOSC)	216
2.16.1. 概述	216
2.16.2. 使用方法	217
2.16.3. 启动延迟	217
2.16.4. XOSC 计数器	217
2.16.5. 休眠模式	218
2.16.6. 程序员模型	218
2.16.7. 寄存器列表	219
2.17. 环形振荡器 (ROSC)	221
2.17.1. 概述	221
2.17.2. ROSC/XOSC 折衷	222
2.17.3. 频率调整	222
2.17.4. ROSC 分频器	223

2.17.5. 随机数生成器	223
2.17.6. ROSC 计数器	223
2.17.7. 休眠模式	223
2.17.8. 寄存器列表	224
2.18. PLL	228
2.18.1. 概述	228
2.18.2. PLL 参数计算	228
2.18.3. 配置	232
2.18.4. 寄存器列表	234
2.19. GPIO	236
2.19.1. 概述	236
2.19.2. 功能选择	237
2.19.3. 中断	239
2.19.4. 引脚	240
2.19.5. 软件示例	241
2.19.6. 寄存器列表	244
2.20. 系统信息	305
2.20.1. 概述	305
2.20.2. 寄存器列表	305
2.21. Syscfg	306
2.21.1. 概述	306
2.21.2. 寄存器列表	306
2.22. TBMAN	309
2.22.1. 寄存器列表	309
3. PIO	311
3.1. 概述	311
3.2. 程序员模型	312
3.2.1. PIO 程序	312
3.2.2. 控制流	313
3.2.3. 寄存器	314
3.2.4. 阻塞	317
3.2.5. 引脚映射	318
3.2.6. IRQ 标志	318
3.2.7. 状态机之间的交互	318
3.3. PIO 汇编器 (pioasm)	319
3.3.1. 指令	319
3.3.2. 值	320
3.3.3. 表达式	320
3.3.4. 注释	320
3.3.5. 标签	320
3.3.6. 指令	321
3.3.7. 伪指令	321
3.4. 指令集	321
3.4.1. 概要	321
3.4.2. JMP	322
3.4.3. WAIT	323
3.4.4. IN	324
3.4.5. OUT	325
3.4.6. PUSH	326
3.4.7. PULL	327
3.4.8. MOV	328
3.4.9. IRQ	329
3.4.10. SET	330
3.5. 功能细节	331
3.5.1. 旁路设定	331
3.5.2. 程序封装	332
3.5.3. FIFO 连接	334
3.5.4. 自动推送与自动拉取	335
3.5.5. 时钟分频器	339
3.5.6. GPIO 映射	340

3.5.7. 强制执行及 EXEC 指令	342
3.6. 示例	344
3.6.1. 双工 SPI	344
3.6.2. WS2812 LED	348
3.6.3. UART 发送	350
3.6.4. UART 接收	352
3.6.5. 曼彻斯特编码串行发送与接收	355
3.6.6. 差分曼彻斯特编码 (BMC) 发送与接收	357
3.6.7. I2C	361
3.6.8. PWM	364
3.6.9. 加法	366
3.6.10. 更多示例	367
3.7. 寄存器列表	368
4. 外设	383
4.1. USB	383
4.1.1. 概述	383
4.1.2. 架构	384
4.1.3. 程序员模型	394
4.1.4. 寄存器列表	398
参考文献	417
4.2. UART	417
4.2.1. 概述	417
4.2.2. 功能描述	418
4.2.3. 操作	420
4.2.4. UART 硬件流控制	422
4.2.5. UART DMA 接口	424
4.2.6. 中断	425
4.2.7. 程序员模型	427
4.2.8. 寄存器列表	429
4.3. I2C	440
4.3.1. 特性	440
4.3.2. IP 配置	441
4.3.3. I2C 概述	441
4.3.4. I2C 术语	443
4.3.5. I2C 行为	444
4.3.6. I2C 协议	445
4.3.7. 发送 FIFO 管理及 START、STOP 和 RESTART 生成	448
4.3.8. 多主机仲裁	450
4.3.9. 时钟同步	451
4.3.10. 操作模式	452
4.3.11. 尖峰抑制	457
4.3.12. 快速模式 Plus 操作	458
4.3.13. 总线清除功能	458
4.3.14. IC_CLK 频率配置	459
4.3.15. DMA 控制器接口	463
4.3.16. 中断寄存器操作	464
4.3.17. 寄存器列表	464
4.4. SPI	501
4.4.1. 概述	502
4.4.2. 功能描述	502
4.4.3. 操作	505
4.4.4. 寄存器列表	515
4.5. PWM	521
4.5.1. 概述	521
4.5.2. 程序员模型	522
4.5.3. 寄存器列表	529
4.6. 定时器	534
4.6.1. 概述	534
4.6.2. 计数器	535
4.6.3. 报警	535

4.6.4. 程序员模型	536
4.6.5. 寄存器列表	539
4.7. 看门狗	544
4.7.1. 概述	544
4.7.2. 时钟节拍生成	544
4.7.3. 看门狗计数器	545
4.7.4. 暂存寄存器	545
4.7.5. 程程序员模型	545
4.7.6. 寄存器列表	547
4.8. 实时时钟 (RTC)	548
4.8.1. 存储格式	548
4.8.2. 闰年	549
4.8.3. 中断	549
4.8.4. 参考时钟	549
4.8.5. 程程序员模型	550
4.8.6. 寄存器列表	553
4.9. 模数转换器 (ADC) 及温度传感器	557
4.9.1. ADC 控制器	558
4.9.2. SAR 型 ADC	559
4.9.3. ADC 有效位数 (ENOB)	561
4.9.4. 积分非线性 (INL) 与差分非线性 (DNL)	562
4.9.5. 温度传感器	563
4.9.6. 寄存器列表	564
4.10. 串行同步接口 (SSI)	567
4.10.1. 概述	568
4.10.2. 特性	568
4.10.3. IP 修改	569
4.10.4. 时钟比率	570
4.10.5. 发送与接收 FIFO 缓冲区	571
4.10.6. 32 位帧尺寸支持	572
4.10.7. SSI 中断	572
4.10.8. 传输模式	573
4.10.9. 操作模式	574
4.10.10. 对端连接接口	579
4.10.11. DMA 控制器接口	595
4.10.12. APB 接口	597
4.10.13. 寄存器列表	598
5. 电气与机械	607
5.1. 封装	607
5.1.1. 热特性	608
5.1.2. 推荐 PCB 布局	608
5.1.3. 封装标记	608
5.2. 存储条件	609
5.3. 焊接曲线	609
5.4. 合规性	611
5.5. 引脚定义	611
5.5.1. 引脚位置	611
5.5.2. 引脚说明	612
5.5.3. 引脚规格	614
5.6. 电源供应	622
5.7. 功耗	622
5.7.1. 外设功耗	622
5.7.2. 典型用户案例功耗	623
附录A：寄存器字段类型	625
标准类型	625
RW:	625
RO:	625
WO:	625
清除类型	625
SC	625

WC	625
FIFO 类型	626
RWF	626
RF	626
WF	626
附录 B：勘误	627
Bootrom	627
RP2040-E9	627
RP2040-E14	627
时钟	628
RP2040-E7	628
RP2040-E10	628
DMA	629
RP2040-E12	629
RP2040-E13	629
GPIO / ADC	630
RP2040-E6	630
RP2040-E11	630
USB	630
RP2040-E2	630
RP2040-E3	631
RP2040-E4	631
RP2040-E5	631
RP2040-E15	633
RP2040-E16	634
看门狗	634
RP2040-E1	634
XIP Flash	635
RP2040-E8	635
附录 C：供应情况	636
支持	636
订购代码	636
文档发行历史	637
2025年2月20日	637
2024年10月15日	637
2024年5月2日	637
2024年2月2日	637
2023年6月14日	637
2023年3月3日	637
2022年12月1日	637
2022年6月30日	638
2022年6月17日	638
2021年11月4日	638
2021年11月3日	638
2021年9月30日	638
2021年6月23日	638
2021年6月7日	639
2021年4月13日	639
2021年4月7日	639
2021年3月5日	639
2021年2月23日	639
2021年2月1日	639
2021年1月26日	639
2021年1月21日	640

第一章 引言

微控制器连接软件世界与硬件世界。它们使开发者能够以与数字逻辑相同的确定性和周期精度，编写与物理世界交互的软件。它们位于性价比空间的左下角，以十倍于性能更强同类产品的销量领先。

它们是推动我们世界数字化转型的中坚力量。

RP2040是Raspberry Pi首款推出的微控制器。它将我们标志性的高性能、低成本和易用性价值带入微控制器领域。

凭借大容量片上存储器、对称双核处理器架构、确定性总线结构及丰富的外设组合，辅以我们独特的可编程I/O（PIO）子系统，为专业用户提供无与伦比的性能与灵活性。通过详尽的文档、成熟的MicroPython移植版本以及内置于ROM中的UF2引导程序，为初学者和爱好者用户提供了最低的入门门槛。

RP2040 为无状态设备，支持从外部 QSPI 存储器进行缓存执行。该设计决策允许您根据应用需求选择合适容量的非易失性存储器，并享受普通 Flash 器件的低价优势。

RP2040 采用先进的 40nm 工艺制造，提供高性能、低动态功耗和低漏电，具备多种低功耗模式，以支持电池供电下的长时间运行。

主要特性：

- 双核 ARM Cortex-M0+，主频 133MHz
- 六个独立储存区，片上 SRAM 总容量为 264kB
- 通过专用 QSPI 总线，支持最多 16MB 的片外 Flash 存储
- DMA 控制器
- 全连接 AHB 交叉开关
- 插值器与整数除法器外设
- 片上可编程低压差稳压器（LDO），用于生成核心电压
- 两个片上相位锁定环（PLL），用于生成 USB 和核心时钟
- 30 个 GPIO 引脚，其中 4 个可用作模拟输入
- 外设
 - 2 个 UART
 - 2 个 SPI 控制器
 - 2 个 I2C 控制器
 - 16 个 PWM 通道
 - USB 1.1 控制器及 PHY，支持主机和设备
 - 8 个 PIO 状态机

无论您的微控制器应用领域为何，从机器学习到电机控制，从农业到音频，RP2040 均具备足够的性能、功能与支持，助力您的产品腾飞。

1.1. 芯片为何命名为 RP2040？

RP2040 后缀数字源自以下内容：

1. 处理器核心数量 (2)
2. 大致的处理器类型 (M0+)
3. $\lfloor \log_2(\text{内存} / 16\text{kB}) \rfloor$
4. $\lfloor \log_2(\text{非易失性存储} / 128\text{kB}) \rfloor$, 无板载非易失性存储时为 0

参见图 1。

图1。RP2040芯片名称说明。



1.2. 概要

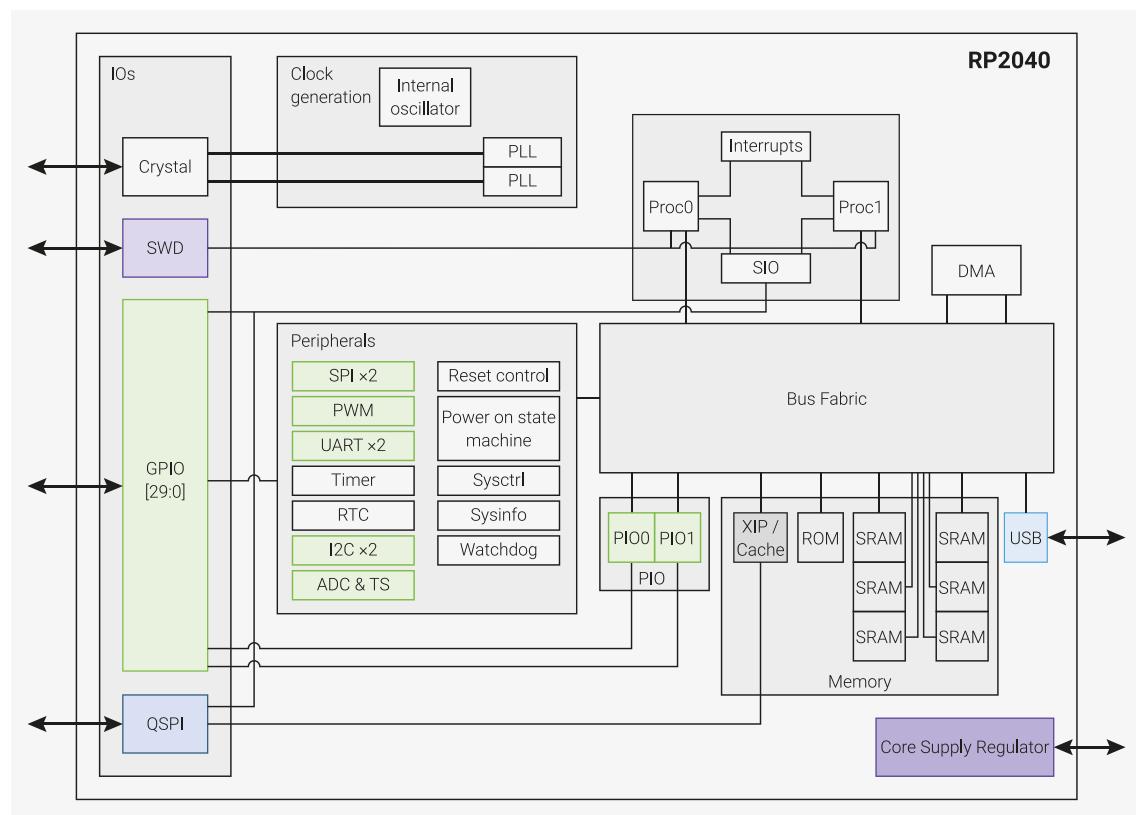
RP2040是一款低成本、高性能的微控制器设备，具备灵活的数字接口。主要特性包括：

- 双核Cortex M0+处理器，最高133MHz（或在1.15V时达200MHz，详见第2.15.3节）
- 264kB嵌入式SRAM，分为6个存储块
- 30个多功能GPIO口
- 6个专用SPI Flash IO端口（支持XIP）
- 为常用外设设计的专用硬件模块
- 可编程IO，支持扩展外设功能
- 4通道ADC，内置温度传感器，500ksps，12位转换精度
- USB 1.1 主机/设备模式

1.3. 芯片

RP2040集成了双核M0+处理器、DMA、内部存储器及通过AHB/APB总线连接的外设模块。

图2。 RP2040芯
片系统概述



代码可通过专用的 SPI、DSPI 或 QSPI 接口直接从外部存储器执行。小缓存可提升典型应用的性能。

可通过 SWD 接口实现调试功能。

内部 SRAM 可用于存储代码或数据。其地址空间为单一 264 kB 区域，但物理上划分为 6 个存储块，以允许不同主控单元的并行访问。

DMA 总线主控可用以分担处理器的重复数据传输任务。

GPIO 引脚可直接驱动，或由多种专用逻辑功能驱动。

设有专用硬件支持的固定功能，如 SPI、I2C、UART。

灵活可配置的 PIO 控制器可用于实现多种 IO 功能。

嵌入式 PHY 的 USB 控制器可在软件控制下提供 FS/LS 主机或设备连接。

四个 ADC 输入端口，与 GPIO 引脚共享。

两个 PLL，用于提供 USB 或 ADC 的固定 48MHz 时钟，以及最高 133MHz 的灵活系统时钟。

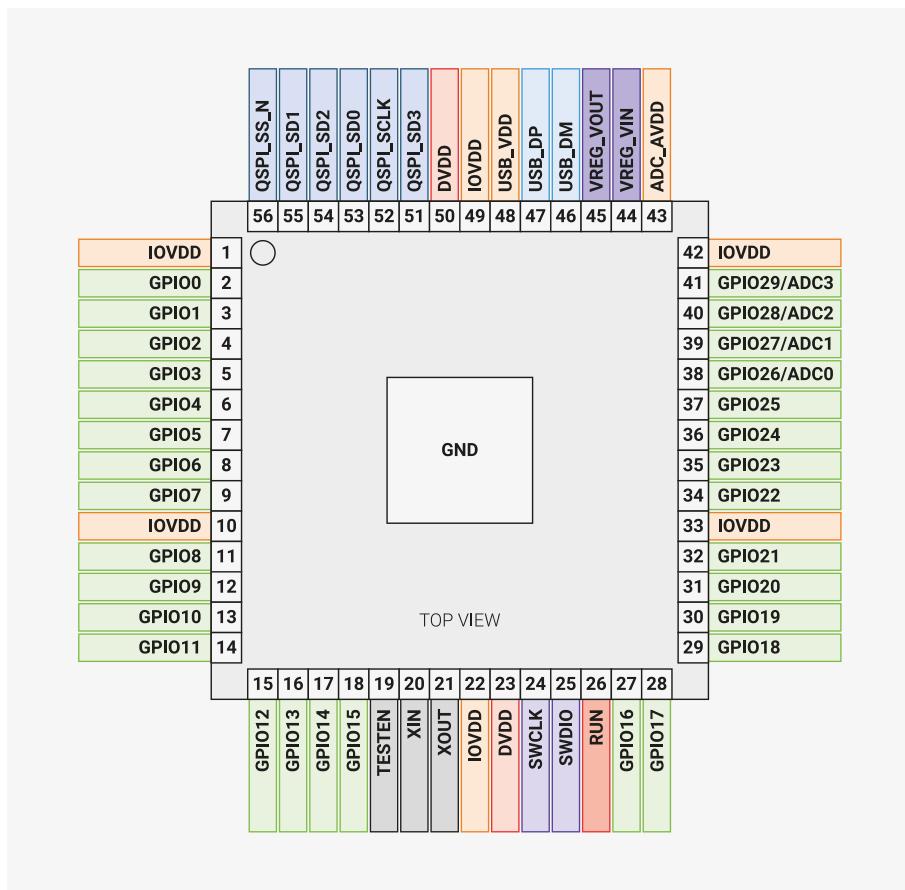
内部电压调节器负责提供核心电压，因此最终产品仅需提供 IO 电压。

1.4. 引脚参考

本节提供引脚排列及引脚功能的快速参考。完整详情，包括电气规格及封装图，详见第5章。

1.4.1. 引脚位置

图3。QFN-56 7x7mm
(减小ePad尺寸)
的RP2040引脚
排列



1.4.2. 引脚描述

表1。此处简要描述各引脚功能。
完整电气规格详见第5章。
。

名称	描述
GPIOx	通用数字输入与输出。RP2040可将多个内部外设之一连接至每个GPIO，或通过软件直接控制GPIO。
GPIOx/ADCy	通用数字输入与输出，具备模数转换功能。RP2040的ADC配备模拟多路复用器，可选择任一引脚进行电压采样。
QSPIx	接口用于SPI、双SPI或四路SPI闪存设备，支持执行在位功能。若不用于闪存访问，此类引脚亦可作为软件控制的GPIO使用。
USB_DM 和 USB_DP	USB控制器，支持全速设备及全速/低速主机。每个引脚须配备27Ω串联终端电阻，但总线的上拉和下拉电阻由内部提供。
XIN 和 XOUT	将晶体连接至RP2040的晶体振荡器。XIN亦可用作单端CMOS时钟输入，需断开XOUT。USB引导加载程序要求12MHz晶体或12MHz时钟输入。推荐晶体详见晶体振荡器（第2.16节）。
RUN	全局异步复位引脚。置低复位，置高运行。若无需外部复位，该引脚可直接连接至IOVDD。
SWCLK 和 SWDIO	访问内部串行线调试多路总线。提供对两处理器的调试访问，亦可用于代码下载。
测试	工厂测试模式引脚，连接至地线(GND)。
GND	单一外部地线连接，连接至RP2040芯片内部多个地线焊盘。
IOVDD	数字GPIO电源，额定电压1.8V至3.3V。

名称	描述
USB_VDD	内部USB全速PHY电源，额定电压3.3V。
ADC_AVDD	模数转换器电源，额定电压3.3V。
VREG_VIN	内部核心稳压器电源输入，额定电压1.8V至3.3V。
VREG_VOUT	内部核心稳压器电源输出，额定电压1.1V，最大电流100mA。
DVDD	数字核心电源，额定电压1.1V。可连接至VREG_VOUT或其他板级电源。

1.4.3. GPIO 功能

每个GPIO引脚均可通过以下定义的GPIO功能连接至内部外设。部分内部外设连接在多个位置出现，以提供系统级灵活性。SIO、PIO0 和 PIO1 可连接至所有 GPIO 引脚，并由软件（或软件控制状态机）管理，因此可用于实现多种功能。

表2。通用输入/输出 (GPIO) 组功能

GPIO	功能								
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB 过流检测
1	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1		USB 总线电压检测
2	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
3	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1		USB 过流检测
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1		USB 总线电压检测
5	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
6	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1		USB 过流检测
7	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1		USB 总线电压检测
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
9	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1		USB 过流检测
10	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线电压检测
11	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 过流检测
13	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线电压检测
14	SPI1 SCK	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM7 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
15	SPI1 TX	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM7 B	SIO	PIO0	PIO1		USB 过流检测
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB 总线电压检测
17	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
18	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1		USB 过流检测
19	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1		USB 总线电压检测
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0	USB 总线使能 (VBUS EN)
21	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB 过流检测

功能									
22	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB 总线电压检测
23	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB 总线使能 (VBUS EN)
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB 过流检测
25	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB 总线电压检测
26	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
27	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 过流检测
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 总线电压检测
29	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)

表 3。GPIO 组 0
功能描述

功能名称	描述
SPIx	将内部 PL022 SPI 外设之一连接至 GPIO
UARTx	将内部 PL011 UART 外设之一连接至 GPIO
I2Cx	将内部 DW I2C 外设之一连接至 GPIO
PWMx A/B	将PWM切片连接至GPIO。共有八个PWM切片，每个切片包含两个输出通道（A/B）。B引脚亦可用作输入，用于频率和占空比的测量。
SIO	通过单周期IO (SIO) 模块实现GPIO的软件控制。必须选择SIO功能 (F5) 以使处理器驱动GPIO，但输入端始终连接，因此软件能够随时检测GPIO状态。
PIOx	将可编程IO模块 (PIO) 之一连接至GPIO。PIO可实现多种接口，且内置引脚映射硬件，允许数字接口灵活地分配至银行0的GPIO。须选择PIO功能 (F6, F7) 以使PIO驱动GPIO，但输入端始终连接，PIO可持续监测所有引脚状态。
CLOCK GPINx	通用时钟输入。可路由至RP2040的多个内部时钟域，例如为RTC提供1Hz时钟，或连接至内部频率计数器。
CLOCK GPOUTx	通用时钟输出。可将多个内部时钟（包括PLL输出）驱动至GPIO，并支持可选的整数分频。
USB OVCUR DET/VBUS DET/VBUS EN	内部USB控制器的USB电源控制信号。

第2章 系统描述

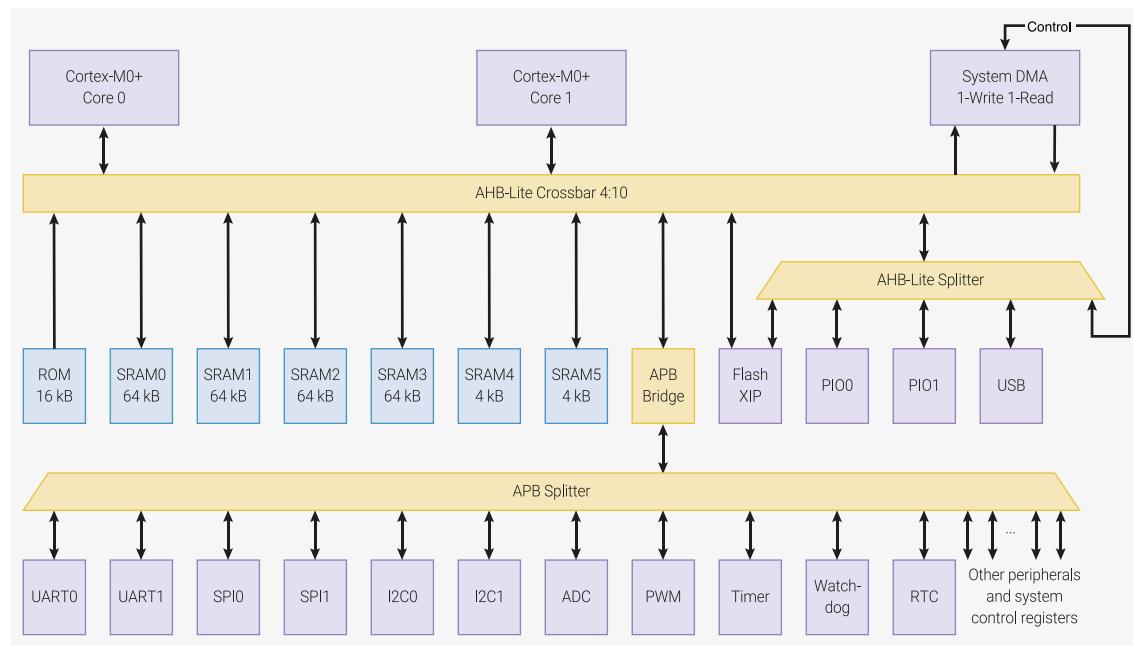
本章介绍RP2040的关键系统特性，包括处理器、内存、模块连接方式、时钟、复位、电源与IO。详见图2的概述图。

2.1. 总线结构

RP2040总线结构负责在芯片内部路由地址和数据信号。

图4展示了总线结构的高级架构。主 AHB-Lite 交叉开关在 4 个上游端口和 10 个下游端口间路由地址与数据：每个时钟周期最多支持四次总线传输。所有数据路径均为 32 位宽。内存设备在主交叉开关上设有专用端口，以满足其高带宽需求。高带宽的 AHB-Lite 外设在交叉开关上共享一个端口，APB 桥提供对系统控制寄存器及低带宽外设的总线访问。

图4。RP2040总线
结构概述



总线结构连接了 4 个 AHB-Lite 主设备，即生成地址的设备：

- 处理器核0
- 处理器核1
- DMA控制器读端口
- DMA控制器写端口

上述设备通过主交叉开关连接至 10 个下游端口：

- ROM
- Flash XIP
- SRAM 0 至 5（各设一个端口）
- 高速 AHB-Lite 外设：PIO0、PIO1、USB、DMA 控制寄存器、XIP 辅助端口（共享一个端口）
- 连接所有 APB 外设及系统控制寄存器的桥接

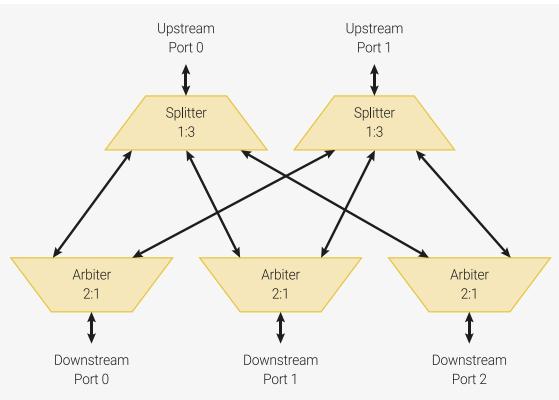
四个总线主设备可同时访问任意四个不同的交叉开关端口，总线结构不引入等待

针对任何AHB-Lite从设备访问的状态。因此，在系统时钟为125MHz时，最大持续总线带宽为2.0GBps。系统地址映射已被设计为使该并行带宽可服务于尽可能多的软件应用场景 —— 例如，条带化SRAM别名（第2.6.2节）将主存访问分散到四个交叉开关端口（[SRAM0...3](#)），以使更多存储访问能够并行进行。

2.1.1. AHB-Lite 交叉开关

RP2040总线结构的核心是一个4:10全连接交叉开关。其4个上游端口连接至4个系统总线主设备，10个下游端口连接至带宽最高的AHB-Lite从设备（即内存接口）及总线结构的下层部分。图5展示了2:3的AHB-Lite交叉开关结构，排列与RP2040上的4:10交叉开关相同，但图中展示更为简洁。

图5。一个2:3的AHB-Lite交叉开关。
每个上游端口连接至一个分配器，该分配器将总线请求引导至三个下游端口中的一个，并回传响应。每个下游端口连接至一个仲裁器，安全管理对该端口的并发访问。



交叉开关由以下两个组件构成：

- 分配器
 - 执行粗地址解码
 - 将请求（地址、写入数据）引导至初始地址解码指示的下游端口
 - 将来自正确仲裁器的响应（读取数据、总线错误）回传至上游端口
- 仲裁器
 - 管理对下游端口的并发请求
 - 将响应（读取数据、总线错误）引导至正确的分配器
 - 实施总线优先级规则

RP2040上的主交叉开关由4个1:10分配器和10个4:1仲裁器组成，彼此之间由40条AHB-Lite总线通道连接。注意，由于AHB-Lite为流水线总线，分配器可能在回传针对下游端口A的先前请求响应时，同时处理对下游端口B的新请求。这不会引起任何周期惩罚。

2.1.1.1. 总线优先级

主AHB-Lite交叉开关中的仲裁器实现了两级总线优先级方案。优先级等级按主控单元配置，使用BUSCTRL寄存器组中的US_PRIORITY寄存器。

当多个访问请求同时到达同一仲裁器时，来自高优先级主控单元（优先级等级为1）的请求将优先于低优先级主控单元（优先级等级为0）的请求。若多个相同优先级的主控单元同时试图访问同一从设备，将采用轮询方式进行平局处理，即仲裁器依次赋予各主控单元访问权限。

注意

优先级仲裁仅适用于多个主控单元在同一周期尝试访问同一从设备的情况。
访问不同的从设备，例如不同的SRAM存储区，可以同时进行。

当访问零等待状态的从设备时，如SRAM（即每个系统时钟周期可访问一次），高优先级主控单元不会因低优先级主控单元的访问而受到任何延迟或其他时序影响。
这允许保证硬实时使用场景下的延迟和吞吐量；但这意味着低优先级主设备可能会被阻塞，直到出现空闲周期。

2.1.1.2. 总线性能计数器

性能计数器自动统计对主AHB-Lite交叉开关仲裁器的访问次数。这有助于诊断高流量使用场景中的性能问题。

共有四个性能计数器。每个计数器为24位饱和计数器。计数器数值可通过`BUSCTRL_PERFCTRx`读取，向`BUSCTRL_PERFCTRx`写入任意值可清零。每个计数器一次可计数20个可选事件中的一个，由`BUSCTRL_PERFSELx`配置。可用总线事件如下：

PERFSEL _x	事件	描述
0	APB访问，存在竞争	完成对APB仲裁器（位于所有APB外设上游）的访问，该访问曾因另一主设备的访问而延迟。
1	APB访问	对APB仲裁器访问的完成
2	FASTPERI访问，存在竞争	对FASTPERI仲裁器的访问完成（此仲裁器位于PIO、DMA配置端口、USB及XIP辅助FIFO端口的上游），此前因其他主控访问而被延迟。
3	FASTPERI访问	对FASTPERI仲裁器访问的完成
4	SRAM5访问，存在竞争	对SRAM5仲裁器的访问完成，此前因其他主控访问而被延迟。
5	SRAM5访问	对SRAM5仲裁器访问的完成
6	SRAM4访问，存在竞争	对SRAM4仲裁器的访问完成，此前因其他主控访问而被延迟。
7	SRAM4访问	对SRAM4仲裁器访问的完成
8	SRAM3访问，存在竞争	对SRAM3仲裁器的访问完成，此前因其他主控访问而被延迟。
9	SRAM3访问	完成对SRAM3仲裁器的访问
10	SRAM2访问，争用中	完成对SRAM2仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
11	SRAM2访问	完成对SRAM2仲裁器的访问
12	SRAM1访问，争用中	完成对SRAM1仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
13	SRAM1访问	完成对SRAM1仲裁器的访问
14	SRAM0访问，争用中	完成对SRAM0仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
15	SRAM0访问	完成对SRAM0仲裁器的访问

PERFSEL x	事件	描述
16	XIP_MAIN 访问， 争用中	完成对 XIP_MAIN 仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
17	XIP_MAIN 访问	完成对 XIP_MAIN 仲裁器的访问
18	ROM访问， 受阻	对ROM仲裁器的访问完成，该访问因另一主控设备的访问而被延迟。
19	ROM访问	对ROM仲裁器的访问完成

2.1.2. 原子寄存器访问

每个外设寄存器块分配了4kB地址空间，寄存器通过以下四种方法之一访问，由地址译码选择。

- 地址 + **0x0000**: 普通读写访问
- 地址 + **0x1000**: 写时原子异或
- 地址 + **0x2000**: 写时原子位掩码置位
- 地址 + **0x3000**: 写时原子位掩码清零

这允许对控制寄存器的单个字段进行修改，无需执行软件中的读-改-写序列；修改直接提交给外设，并现场执行。如果没有此功能，当中断服务例程与前台代码并发运行，或两个处理器并行执行代码时，安全访问IO寄存器将非常困难。

四个原子访问别名共占用16kB空间。RP2040上的大多数外设本身具备此功能，且原子写入的时序与普通读写访问相同。部分外设（I2C、UART、SPI和SSI）通过总线插入设备实现此功能，该设备在外设边界将上游的原子写操作转换为下游的读-改-写序列，从而使访问时间增加两个系统时钟周期。

SIO（第2.3.1节）为直接连接至核IO端口的单周期IO模块，不支持总线级别的原子访问，尽管部分寄存器（例如GPIO）提供设定／清除／异或别名。

2.1.3. APB 桥接

APB桥接器将高速主AHB-Lite互连与低带宽外设连接。虽然AHB-Lite结构在各处均支持零等待态访问，但APB访问存在周期惩罚：

- APB总线访问至少耗时两个周期（设置阶段和访问阶段）。
- 由于总线请求和响应采取寄存方式，桥接器在读访问中增加了一个额外周期。
- 由于APB设置阶段必须等到AHB-Lite写数据有效后才能开始，桥接器在写访问中增加了两个额外周期。

因此，APB部分总线结构的吞吐量略低于AHB-Lite部分。然而，带宽完全足以满足APB串行外设的饱和需求。

2.1.4. 窄带 IO 寄存器写入

RP2040上的内存映射IO寄存器忽略总线读/写访问的宽度。它们将所有写操作视为32位大小。这意味着软件不能使用字节或半字写操作来修改IO寄存器的部分内容：任何地址的30个高位地址与寄存器地址匹配的写操作都会影响整个

寄存器的内容。

在不使用读-改-写序列的情况下，更新IO寄存器部分的最佳方案是RP2040上的原子设置/清除/XOR操作（参见第2.1.2节）。请注意，这比字节或半字写操作更灵活，因为任何字段组合都可以在一次操作中更新。

在执行8位或16位写操作（例如Cortex-M0+上的 `strb` 指令）时，IO寄存器将采样完整的32位写入数据总线。RP2040上的 Cortex-M0+和DMA总是会在总线上复制窄宽数据：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/system/narrow_io_write/narrow_io_write.c 第19至62行

```

19 int main() {
20     stdio_init_all();
21
22     // 我们将使用WATCHDOG_SCRATCH0作为方便的32位读写寄存器
23     // 可向其赋予任意值
24     io_rw_32 *scratch32 = &watchdog_hw->scratch[0];
25     // 将scratch寄存器别名为偏移+0x0和+0x2处的两个半字
26     volatile uint16_t *scratch16 = (volatile uint16_t *) scratch32;
27     // 将暂存寄存器别名为偏移量+0x0、+0x1、+0x2、+0x3处的四个字节：
28     volatile uint8_t *scratch8 = (volatile uint8_t *) scratch32;
29
30     // 演示我们可以像正常操作一样读写暂存寄存器：
31     printf("正在写入32位数值\n");
32     *scratch32 = 0xdeadbeef;
33     printf("应为0xdeadbeef: 0x%08x\n", *scratch32);
34
35     // 我们可以正常进行窄宽度读取——IO寄存器将此视为一次32位
36     // 读取，处理器/DMA将根据传输大小和地址最低有效位选择正确的字节通道
37     // on transfer size and address LSBs
38 printf("\n逐字节读取返回\n");
39     // 小端序！
40     printf("应为 ef be ad de: %02x ", scratch8[0]);
41     printf("%02x ", scratch8[1]);
42     printf("%02x ", scratch8[2]);
43     printf("%02x\n", scratch8[3]);
44
45     // 字节写入在32位总线上复制四次，IO
46     // 寄存器通常采样整个写入总线。
47     printf("\n在偏移量0处写入8位值0xa5\n");
48     scratch8[0] = 0xa5;
49     // 一次性读取整个scratch寄存器
50     printf("应为0xa5a5a5a5: 0x%08x\n", *scratch32);
51
52     // IO寄存器忽略地址最低有效位[1:0]及传输
53     // 大小，因此字节偏移无关紧要
54     printf("\n在偏移量1处写入8位值\n");
55     scratch8[1] = 0x3c;
56     printf("应为 0x3c3c3c3c: 0x%08x\n", *scratch32);
57
58     // 半字写入操作也在写数据总线上进行复制
59     printf("\n在偏移量0处写入16位值\n");
60     scratch16[0] = 0xf00d;
61     printf("应为 0xf00df00d: 0x%08x\n", *scratch32);
62 }
```

2.1.5. 寄存器列表

总线结构寄存器的基址起始于 `0x40030000`（在 SDK 中定义为 `BUSCTRL_BASE`）。

表 4. BUSCTRL 寄存器列表

偏移量	名称	说明
0x00	BUS_PRIORITY	设置每个主设备的总线仲裁优先级。
0x04	BUS_PRIORITY_ACK	总线优先级确认
0x08	PERFCTR0	总线结构性能计数器 0
0x0c	PERFSEL0	PERFCTR0 的总线结构性能事件选择
0x10	PERFCTR1	总线结构性能计数器 1
0x14	PERFSEL1	PERFCTR1 的总线结构性能事件选择
0x18	PERFCTR2	总线结构性能计数器 2
0x1c	PERFSEL2	PERFCTR2 的总线结构性能事件选择
0x20	PERFCTR3	总线结构性能计数器 3
0x24	PERFSEL3	PERFCTR3 的总线结构性能事件选择

BUSCTRL: BUS_PRIORITY 寄存器

偏移: 0x00

描述

设置每个主设备的总线仲裁优先级。

表 5. BUS_PRIORITY 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	DMA_W : 0 - 低优先级, 1 - 高优先级	读写	0x0
11:9	保留。	-	-
8	DMA_R : 0 - 低优先级, 1 - 高优先级	读写	0x0
7:5	保留。	-	-
4	PROC1 : 0 - 低优先级, 1 - 高优先级	读写	0x0
3:1	保留。	-	-
0	PROC0 : 0 - 低优先级, 1 - 高优先级	读写	0x0

BUSCTRL: BUS_PRIORITY_ACK 寄存器

偏移: 0x04

描述

总线优先级确认

表 6. BUS_PRIORITY_ACK 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	所有仲裁器注册新的全局优先级后, 值变为 1。 仲裁器在处理中断的非连续访问时更新其本地优先级。 在正常情况下, 此过程几乎立即发生。	只读	0x0

BUSCTRL: PERFCTR0 寄存器

偏移: 0x08

描述

总线结构性能计数器 0

表 7. PERFCTR0
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构性能计数器 0 对总线结构仲裁器的某些事件信号进行计数。 写入任意值以清除。通过 PERFSEL0 选择欲计数的事件	WC	0x000000

BUSCTRL: PERFSEL0 寄存器

偏移: 0x0c

说明

PERFCTR0 的总线结构性能事件选择

表 8. PERFSEL0
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR0 选择事件。可计数主交叉开关下游端口上的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

BUSCTRL: PERFCTR1 寄存器

偏移: 0x10

说明

总线结构性能计数器 1

表9. PERFCTR1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器1，用于计数来自总线结构仲裁器的特定事件信号。 写入任意值以清除。使用 PERFSEL1 选择要计数的事件	WC	0x000000

BUSCTRL: PERFSEL1 寄存器

偏移: 0x14

说明

PERFCTR1 的总线结构性能事件选择

表10. PERFSEL1
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR1 选择一个事件。计数主交叉开关下游端口的竞争访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

BUSCTRL：PERFCTR2 寄存器

偏移：0x18

说明

总线结构性能计数器 2

表11. PERFCTR2 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器2 对总线结构仲裁器的某些事件信号进行计数。 写入任意值以清除。使用 PERFSEL2 选择要计数的事件	WC	0x000000

BUSCTRL：PERFSEL2 寄存器

偏移：0x1c

说明

PERFCTR2 的总线结构性能事件选择

表12. PERFSEL2 寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR2 选择一个事件，统计主交叉开关下游端口的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		

位	描述	类型	复位值
	0x13 → ROM		

BUSCTRL：PERFCTR3 寄存器

偏移: 0x20

说明

总线结构性能计数器 3

表13. PERFCTR3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器3，统计来自总线结构仲裁器的某些事件信号。 写入任意值以清除。使用 PERFSEL3 选择统计的事件。	WC	0x000000

BUSCTRL：PERFSEL3 寄存器

偏移: 0x24

说明

PERFCTR3的总线结构性能事件选择

表14. PERFSEL3 寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR3 选择一个事件，统计主交叉开关下游端口的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		

位	描述	类型	复位值
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

2.2. 地址映射

设备的地址映射划分为若干部分，如表15所示。详细内容见下文章节。
访问未映射的地址范围时将引发总线错误。

2.2.1. 概述

表15. 地址映射
摘要

ROM	0x00000000
XIP	0x10000000
SRAM	0x20000000
APB 外设	0x40000000
AHB-Lite 外设	0x50000000
IOPORT 寄存器	0xd0000000
Cortex-M0+ 内部寄存器	0xe0000000

2.2.2. 详细说明

只读存储器：

ROM_BASE	0x00000000
----------	------------

XIP:

XIP_BASE	0x10000000
XIP_NOALLOC_BASE	0x11000000
XIP_NOCACHE_BASE	0x12000000
XIP_NOCACHE_NOALLOC_BASE	0x13000000
XIP_CTRL_BASE	0x14000000
XIP_SRAM_BASE	0x15000000
XIP_SRAM_END	0x15004000
XIP_SSI_BASE	0x18000000

SRAM, SRAM0-3 交错：

SRAM_BASE	0x20000000
SRAM_STRIPED_BASE	0x20000000

SRAM_STRIPED_END	0x20040000
------------------	------------

SRAM4-5 始终为非交错:

SRAM4_BASE	0x20040000
SRAM5_BASE	0x20041000
SRAM_END	0x20042000

SRAM0-3 的非交错别名:

SRAM0_BASE	0x21000000
SRAM1_BASE	0x21010000
SRAM2_BASE	0x21020000
SRAM3_BASE	0x21030000

APB 外设:

SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40004000
CLOCKS_BASE	0x40008000
RESETS_BASE	0x4000c000
PSM_BASE	0x40010000
IO_BANK0_BASE	0x40014000
IO_QSPI_BASE	0x40018000
PADS_BANK0_BASE	0x4001c000
PADS_QSPI_BASE	0x40020000
XOSC_BASE	0x40024000
PLL_SYS_BASE	0x40028000
PLL_USB_BASE	0x4002c000
BUSCTRL_BASE	0x40030000
UART0_BASE	0x40034000
UART1_BASE	0x40038000
SPI0_BASE	0x4003c000
SPI1_BASE	0x40040000
I2C0_BASE	0x40044000
I2C1_BASE	0x40048000
ADC_BASE	0x4004c000
PWM_BASE	0x40050000
TIMER_BASE	0x40054000
WATCHDOG_BASE	0x40058000
RTC_BASE	0x4005c000

ROSC_BASE	0x40060000
VREG_AND_CHIP_RESET_BASE	0x40064000
TBMAN_BASE	0x4006c000

AHB-Lite 外设：

DMA_BASE	0x50000000
----------	------------

USB 基址处设有 DPRAM，紧接其后为寄存器：

USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000

其余 AHB-Lite 外设：

PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
XIP_AUX_BASE	0x50400000

IOPORT 外设：

SIO_BASE	0xd0000000
----------	------------

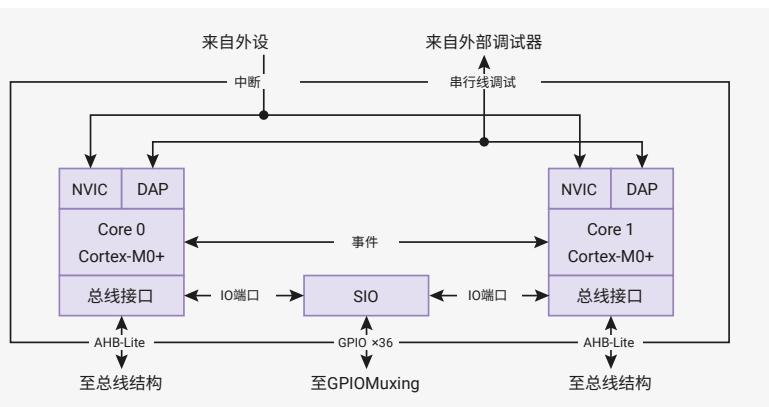
Cortex-M0+ 内部外设：

PPB_BASE	0xe0000000
----------	------------

2.3. 处理器子系统

RP2040 处理器子系统由两个 Arm Cortex-M0+ 处理器组成，每个处理器均配备标准的内部 Arm CPU 外设，以及用于 GPIO 访问和核间通信的外部外设。有关 Arm Cortex-M0+ 处理器的详细信息，包括 RP2040 上使用的特定功能配置，请参见第 2.4 节。

图6. 两个Cortex-M0+处理器，每个均配备专用的32位AHB-Lite总线端口，用于代码获取、加载及存储。SIO连接至每个处理器的单周期IOPORT总线，提供GPIO访问、双向通信及其他核心本地外设功能。两个处理器均可通过单一的多点Serial Wire Debug总线进行调试。26个中断（加上NMI）分别路由至每个处理器的NVIC和WIC。



注意

术语 core0 和 core1、proc0 和 proc1 在 RP2040 的寄存器及文档中交替使用，分别指代处理器0和处理器1。

处理器使用多种接口与系统的其他部分进行通信：

- 每个处理器使用其独立的32位AHB-Lite总线访问内存和内存映射外设。
(详见第2.1节)
- 单周期IO模块通过每个处理器的IOPORT提供对GPIO的高速、确定性访问。
- 26个系统级中断被路由至两个处理器。
- 多点Serial Wire Debug总线为外部调试主机提供对两个处理器的调试访问。

2.3.1. SIO

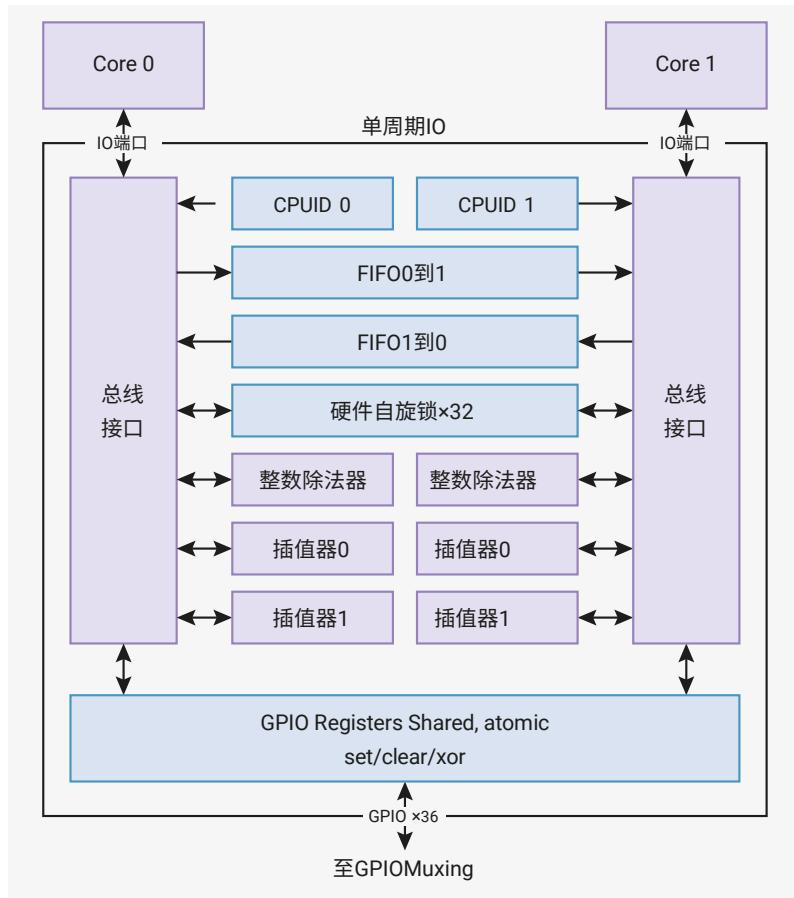
单周期IO模块（SIO）包含多个需要低延迟且确定性访问的外设。它通过每个处理器的IOPORT访问：这是 Cortex-M0+ 上的辅助总线端口，能够执行快速的 32 位读写。SIO 为每个处理器的 IOPORT 提供了专用的总线接口，如图7所示。处理器通过指向特殊 IOPORT 地址段 `0xd` 的正常加载和存储指令访问其 IOPORT。`0000000…0xffffffff`。SIO 作为内存映射硬件，在 IOPORT 地址空间内呈现。

注意

鉴于其严格的时序要求，SIO 并未连接至主系统总线。仅限处理器访问，或通过处理器调试端口由调试器访问。

图7. 单周期IO模块包含处理器必须能够快速访问的内存映射硬件。FIFO和自旋锁支持两个核心之间的消息传递与同步。共享GPIO寄存器提供对支持GPIO的引脚的快速且并发安全的直接访问。

部分内核本地算术硬件可用于加速处理器上的常见任务。



所有IOPORT的读写操作（因此所有SIO访问）均在恰好一个周期内完成，这与Cortex-M0+的主AHB-Lite系统总线不同，后者的加载或存储操作需要两个周期，且可能因其他系统总线主控器的争用而等待更久。这对诸如GPIO等具有严格时序要求的接口至关重要。

SIO寄存器映射至范围为`0xd`的字对齐地址`00000000…0xd000017c`。IOPORT空间的其余部分保留以供将来使用。

SIO外设将在下列章节中作更详细的描述。

2.3.1.1. CPUID

寄存器CPUID是IOPORT空间中的第一个寄存器。核心0访问此地址时读取的值为0，核心1读取的值为1。这是软件确定其运行核心的便捷方法。此项检查在初始引导序列中进行：两个核心同时启动，核心1进入深度睡眠状态，核心0继续执行主引导序列。

！重要

CPUID不应与每个处理器内部私有外设总线上的Cortex-M0+ CPUID寄存器（第2.4.4.1.1节）混淆，后者列示了处理器的部件号及版本。

2.3.1.2. GPIO控制

处理器可通过GPIO寄存器对具GPIO功能的引脚进行快速且直接的控制。存在两套相同的寄存器：

- **GPIO_x** 用于直接控制IO银行0（用户GPIO 0至29，从最低有效位起）
- **GPIO_HI_x** 用于直接控制QSPI IO银行（按顺序为SCLK、SSn、SD0、SD1、SD2、SD3，从最低有效位起）

注意

要通过SIO的GPIO寄存器驱动引脚，必须首先将该引脚的GPIO多路复用器配置为选择SIO GPIO功能。详见表279。

这些GPIO寄存器在双核之间共享，且两核可同时访问。每个银行包含三个寄存器：

- 输出寄存器GPIO_OUT和GPIO_HI_OUT用于设置GPIO的输出电平（1表示高电平，0表示低电平）
- 输出使能寄存器GPIO_OE和GPIO_HI_OE用于使能输出驱动。0表示高阻态，1表示依据GPIO_OUT和GPIO_HI_OUT驱动高/低电平。
- 输入寄存器GPIO_IN 和 GPIO_HI_IN 允许处理器采样 GPIO 的当前状态。

读取 GPIO_IN 一次即可返回全部 30 个 GPIO 的值（GPIO_HI_IN 为 6 个）。软件随后可屏蔽不感兴趣的引脚，仅保留所需引脚。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h, 第 859 至 869 行

```

859 static inline bool gpio_get(uint gpio) {
860 #ifdef NUM_BANK0_GPIOS <= 32
861     return sio_hw->gpio_in & (1u << gpio);
862 #else
863     if (gpio < 32) {
864         return sio_hw->gpio_in & (1u << gpio);
865     } else {
866         return sio_hw->gpio_hi_in & (1u << (gpio - 32));
867     }
868 #endif
869 }
```

OUT 和 OE 寄存器同样具备原子性的 SET、CLR 和 XOR 别名，从而允许软件通过一次操作更新部分引脚。这不仅对于两个核心之间安全的并行 GPIO 访问至关重要，也保证了中断处理程序与在单核上运行的前台代码对 GPIO 的安全并发访问。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h 第908至914行

```

908 static inline void gpio_set_mask(uint32_t mask) {
909 #ifdef PICO_USE_GPIO_COPROCESSOR
910     gpioc_lo_out_set(mask);
911 #else
912     sio_hw->gpio_set = mask;
913 #endif
914 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h 第955至961行

```

955 static inline void gpio_clr_mask(uint32_t mask) {
956 #ifdef PICO_USE_GPIO_COPROCESSOR
957     gpioc_lo_out_clr(mask);
958 #else
959     sio_hw->gpio_clr = mask;
960 #endif
961 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h, 第1145至1170行

```

1145 static inline void gpio_put(uint gpio, bool value) {
1146 #ifdef PICO_USE_GPIO_COPROCESSOR
1147     gpioc_bit_out_put(gpio, value);
1148 #elif NUM_BANK0_GPIOS <= 32
1149     uint32_t mask = 1ul << gpio;
1150     if (value)
1151         gpio_set_mask(mask);
1152     else
1153         gpio_clr_mask(mask);
1154 #else
1155     uint32_t mask = 1ul << (gpio & 0x1fu);
1156     if (gpio < 32) {
1157         if (value) {
1158             sio_hw->gpio_set = mask;
1159         } else {
1160             sio_hw->gpio_clr = mask;
1161         }
1162     } else {
1163         if (value) {
1164             sio_hw->gpio_hi_set = mask;
1165         } else {
1166             sio_hw->gpio_hi_clr = mask;
1167         }
1168     }
1169 #endif
1170 }
```

如果两个处理器在同一时钟周期内同时写入 OUT 或 OE 寄存器（或其任一SET/CLR/XOR别名），结果相当于核心0先写入，随后核心1紧接着写入。例如，若核心0对某位执行SET操作，而核心1同时对该位执行XOR操作，则该位将被设置为0，且与其原始值无关。

① 注意

这是两个核心同时写入GPIO寄存器时所产生结果的概念模型。

该寄存器在任何时刻实际上并不包含该中间值。在前述示例中，若引脚初值为0，且核心0执行SET而核心1执行XOR，则GPIO输出保持低电平，不产生任何正向毛刺。

2.3.1.3 硬件自旋锁

SIO 提供 32 个硬件自旋锁，用于管理对共享软件资源的互斥访问。每个自旋锁均为一位标志，映射至不同地址（自 SPINLOCK0 至 SPINLOCK31）。软件通过以下操作之一与自旋锁交互：

- 读取：尝试获取锁。读取值非零则表示锁已成功获取，读取值为零则表示锁已被先前的读取操作获取。

- 写入（任意值）：释放锁。下一次尝试获取该锁将成功。

若两个核心在同一时钟周期尝试获取同一锁，则核心0优先成功。

一般情况下，软件通过重复轮询锁位（即“自旋”）直到成功获取锁。若锁长时间被持有，此方法效率较低，故自旋锁通常用于保护互斥锁、信号量和队列等高级原语中的短临界区。

出于调试目的，可以通过 SPINLOCK_ST 查看所有 32 个自旋锁的当前状态。

2.3.1.4. 处理器间 FIFO (邮件箱)

SIO 包含两个 FIFO，用于在两个核心之间传递数据、消息或有序事件。每个 FIFO 宽度为 32 位，深度为八个条目。其中一个 FIFO 仅能由核心 0 写入，由核心 1 读取。另一个 FIFO 仅能由核心 1 写入，由核心 0 读取。

每个核心通过写入 FIFO_WR 向其输出 FIFO 写入数据，通过读取 FIFO_RD 读取其输入 FIFO。

状态寄存器 FIFO_ST 提供以下状态信号：

- 输入 FIFO 中含有数据 (VLD)
- 输出 FIFO 有空间可用 (RDY)
- 输入 FIFO 曾在空时被读取 (ROE)
- 输出 FIFO 曾在满时被写入 (WOF)

在发送 FIFO 已满时写入，或者在接收 FIFO 为空时读取，不会改变 FIFO 的状态。

FIFO 的当前内容及其级别状态将被保留。然而，这确实可能导致访问 FIFO 的软件发生数据丢失或接收无效数据，因此将置位一个粘性错误标志 (ROE 或 WOF)。

SIO 为每个核心提供一个 FIFO 中断输出，映射至系统中断号 15 和 16。每个中断输出端是对应核心 FIFO_ST 寄存器中 VLD、RDY 及 WOF 位的逻辑或：当任一位为高电平时，中断信号被置位；当三者均为低电平时，中断信号清除。通过向 FIFO_ST 写入任意数值可以清除 ROE 和 WOF 标志，通过从 FIFO 读取数据直至其为空可以清除 VLD 标志。

若相应中断线在 Cortex-M0+ 的 NVIC 中被使能，则每当其 FIFO 中有数据出现，或发生无效的 FIFO 操作（空读或满写）时，处理器将响应中断。通常，核心 0 将使用 IRQ15，核心 1 将使用 IRQ16。如果 IRQs 使用顺序相反，被中断的核心很难正确认识中断原因，因为该核心无法访问另一个核心的 FIFO 状态寄存器。

注意

只有当软件异常时，ROE 和 WOF 才会被置位。通常，当 FIFO 中出现数据（触发 VLD 标志）时，中断处理程序会被触发，并通过从 FIFO 读取数据直至 VLD 恢复为低电平来清除 IRQ。

启动只读存储器（第 2.8 节）中的 `wait_for_vector` 例程使用了核间 FIFO 和 Cortex-M0+ 事件信号，其中核心 1 保持睡眠状态直至被唤醒，并通过 FIFO 提供其初始堆栈指针、入口点和向量表。

2.3.1.5 整数除法器

SIO 为每个核心提供一套 8 周期的有符号/无符号除法及取模模块。计算通过向两个参数寄存器 DIVIDEND 和 DIVISOR 写入被除数和除数来启动。除法器在接下来的 8 个周期内计算该除法的商 / 和余数 %，第 9 个周期可从两个结果寄存器 DIV_QUOTIENT 和 DIV_REMAINDER 读取结果。可通过轮询寄存器 DIV_CSR 中的“就绪”位等待计算完成，或软件插入固定 8 周期延迟。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S 第 12 至 16 行

```

12 regular_func_with_section hw_divider_divmod_s32
13     ldr r3, =(SIO_BASE)
14     str r0, [r3, #SIO_DIV_SDIVIDEND_OFFSET]
15     str r1, [r3, #SIO_DIV_SDIVISOR_OFFSET]
16     b hw_divider_divmod_return

```

注意

软件在这8个周期内可自由执行其他非除法器操作。

操作数寄存器有两种别名：写入有符号别名（DIV_SDIVIDEND 与 DIV_SDIVISOR）将启动有符号计算，写入无符号别名（DIV_UDIVIDEND 与 DIV_UDIVISOR）将启动无符号计算。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S 第20至24行

```
20 regular_func_with_section hw_divider_divmod_u32
21     ldr r3, =(SIO_BASE)
22     str r0, [r3, #SIO_DIV_UDIVIDEND_OFFSET]
23     str r1, [r3, #SIO_DIV_UDIVISOR_OFFSET]
24     b hw_divider_divmod_return
```

注意

每次对操作数寄存器写入时，都会立即开始新的计算，且新的操作数写入会立即终止当前正在进行的任何计算。例如，在用相同除数除多个数时，仅需写入 xDIVISOR，且每次计算的符号由是否写入 SDIVIDEND 或 UDIVIDEND 决定。

为支持中断处理程序入口/出口（或如 RTOS 上下文切换）时的保存与恢复，结果寄存器同样可写。写入结果寄存器将取消当时正在进行的任何操作。DIV_CSR.DIRTY 标志有助于提升保存/恢复效率：当任何除法器寄存器（操作数或结果）被写入时，该标志置位；读取时则清除该标志。

注意

启用后，默认除法器 AEABI 支持将 C 语言中的除法符号 / 和取余符号 % 映射至硬件除法器。使用 SDK 构建软件并直接使用除法器时，务必最后读取商寄存器。这保证了部分除法器状态能够被任何使用除法器的中断代码正确保存和恢复。无论是否需要该值，均应读取商寄存器。

SDK模块https://github.com/raspberrypi/pico-sdk/blob/master/src/common/pico_divider_headers/include/pico/divider.h

 提供了对32位和64位整数除法中钩载C语言/和 % 运算符所需的AEABI实现，以及一些能够同时返回商和余数的附加C函数。所有这些函数在硬件除法器状态“脏”时，均能正确保存并恢复该状态，以便可用于用户代码或中断处理程序代码中。

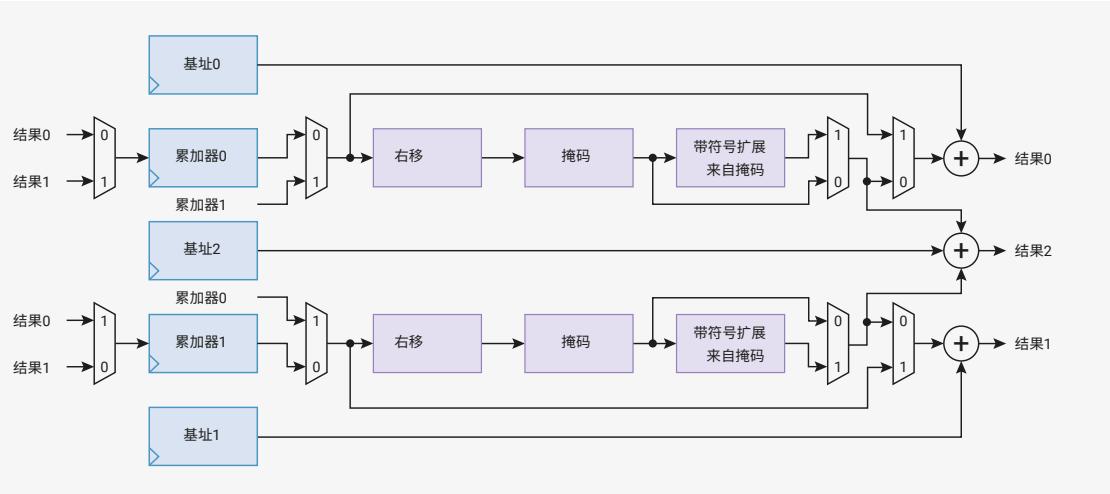
SDK模块hardware_dividerhttps://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/include/hardware/divider.h 提供了访问硬件除法器的底层宏和辅助函数，但这些函数不保存或恢复硬件除法器状态（尽管该头文件确实提供了单独的函数来实现此功能）。

2.3.1.6. 插值器

每个核心配备了两个插值器（INTERP0 和 INTERP1），能够通过将特定预配置操作合并为单个处理器周期以加速任务。适用于预配置操作重复多次的场景，从而使代码在时间关键段使用更少的CPU周期和寄存器。

插值器用于加速SDK中的音频操作，其灵活的配置亦使得量化与抖动、查表地址生成、仿射纹理映射、解压缩及线性反馈等多种任务得到优化。

图8. 一个插值器。两个累加器寄存器及三个基址寄存器均支持处理器单周期读写访问。插值器由两个通道组成，分别对两个累加器执行掩码、移位及符号扩展操作。通过将中间移位/掩码值加至三个基址寄存器，生成三种可能结果。从左至右，每个通道的多路复用器由CTR寄存器中的以下标志位控制：



处理器能在一个周期内读写任意插值器寄存器，结果于下一周期可用。处理器亦可通过写入对应`ACCUMx_ADD`寄存器，对两个累加器`ACCUM0`或`ACCUM1`之一执行加法运算。

`CROSS_RESULT`、
`CROSS_INPUT`、
`SIGNED`、`ADD_RAW`

这三种结果在只读寄存器`PEEK0`、`PEEK1`和`PEEK2`中可用。从上述寄存器读取不会改变插值器的状态。结果同样在寄存器`POP0`、`POP1`、`POP2`处通过别名可访问；从`POPX`别名读取返回的结果与相应的`PEEKX`相同，同时将该通道结果写回累加器。每次读取结果时，可用此方法推进插值器的状态。

此外，插值器支持两个值之间的简单分数混合，以及将值限制在指定范围内。

以下示例展示了弹出一个通道结果以实现简单迭代反馈的基础示例。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第11至23行

```

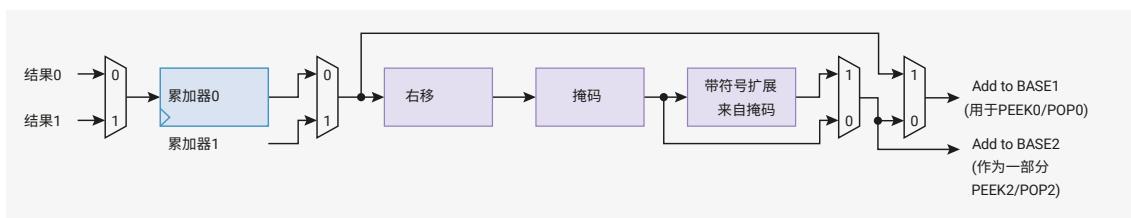
11 void times_table() {
12     puts("9的乘法表：");
13
14     // 在此核心初始化 interp0 通道 0
15     interp_config cfg = interp_default_config();
16     interp_set_config(interp0, 0, &cfg);
17
18     interp0->accum[0] = 0;
19     interp0->base[0] = 9;
20
21     for (int i = 0; i < 10; ++i)
22         printf("%d\n", interp0->pop[0]);
23 }
```

注意

纯属巧合的是，插值器极其适合用于模拟SNES MODE7风格的图形例程。例如，在每个核心上，INTERP0可提供某个仿射变换的图块查找流，而INTERP1则可为相同变换提供图块偏移。

2.3.1.6.1. 通道操作

图9。每个插值器的每个通道均可配置为对指定累加器执行掩码、移位及符号扩展操作。该结果被送入加法器生成最终结果，且该结果可选择性地在每次读取时反馈回累加器。数据通路可通过少量32位多路复用器进行配置。从左至右，这些由以下CTRL标志控制：



每条通路按顺序执行以下三项操作：

- 右移`CTRL_LANEx_SHIFT`位（0至31位）
- 对从`CTRL_LANEx_MASK_LSB`至`CTRL_LANEx_MASK_MSB`（含）范围内的位进行掩码操作（各位均介于0至31位）
- 从掩码最高位进行符号扩展，即若设置了`CTRL_LANEx_SIGNED`，则取位`CTRL_LANEx_MASK_MSB`并将其通过或运算扩展到所有更高有效位

例如，若：

- `ACCUM0 = 0xdeadbeef`
- `CTRL_LANE0_SHIFT = 8`
- `CTRL_LANE0_MASK_LSB = 4`
- `CTRL_LANE0_MASK_MSB = 7`
- `CTRL_SIGNED = 1`

则通路0在每个阶段将产生以下结果：

- 右移8位得到`0x00deadbe`
- 对位7至4加掩码得到`0x00deadbe & 0x000000f0 = 0x000000b0`
- 从位7进行符号扩展得到`0xffffffffb0`

在软件中：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第25至46行

```

25 void moving_mask() {
26     interp_config cfg = interp_default_config();
27     interp0->accum[0] = 0x1234abcd;
28
29     puts("掩码：");
30     printf("ACCUM0 = %08x\n", interp0->accum[0]);
31     for (int i = 0; i < 8; ++i) {
32         // 先最低有效位，再最高有效位。范围包含端点，因此0, 31表示“整个32位寄存器”
33         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
34         interp_set_config(interp0, 0, &cfg);
35         // 从 ACCUMx_ADD 读取返回原始通道移位和掩码值，未加 BASEx
36
36         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
37     }
38
39     puts("带符号扩展的掩码：");
40     interp_config_set_signed(&cfg, true);
41     for (int i = 0; i < 8; ++i) {
42         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
43         interp_set_config(interp0, 0, &cfg);
44         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
45     }
46 }
```

上述示例应打印：

```

ACCUM0 = 1234abcd
Nibble 0: 0000000d
Nibble 1: 000000c0
Nibble 2: 00000b00
Nibble 3: 0000a000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000
带符号扩展的掩码:
Nibble 0: ffffffd
Nibble 1: ffffffc0
Nibble 2: ffffffb00
Nibble 3: fffffa000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000

```

更改结果和输入多路复用器能够在累加器之间形成反馈，此功能在音频抖动处理中尤为有用。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp.c 第48至66行

```

48 void cross_lanes() {
49     interp_config cfg = interp_default_config();
50     interp_config_set_cross_result(&cfg, true);
51     // ACCUM0 获取通道1的结果:
52     interp_set_config(interp0, 0, &cfg);
53     // ACCUM1 获取通道0的结果:
54     interp_set_config(interp0, 1, &cfg);
55
56     interp0->accum[0] = 123;
57     interp0->accum[1] = 456;
58     interp0->base[0] = 1;
59     interp0->base[1] = 0;
60     puts("通道结果交叉: ");
61     for (int i = 0; i < 10; ++i) {
62         uint32_t peek0 = interp0->peek[0];
63         uint32_t pop1 = interp0->pop[1];
64         printf("PEEK0, POP1: %d, %d\n", peek0, pop1);
65     }
66 }

```

应打印：

```

PEEK0, POP1: 124, 456
PEEK0, POP1: 457, 124
PEEK0, POP1: 125, 457
PEEK0, POP1: 458, 125
PEEK0, POP1: 126, 458
PEEK0, POP1: 459, 126
PEEK0, POP1: 127, 459
PEEK0, POP1: 460, 127
PEEK0, POP1: 128, 460
PEEK0, POP1: 461, 128

```

2.3.1.6.2 混合模式

混合模式可在每个核心的 `INTERP0` 上使用，且通过 `CTRL_LANE0_BLEND` 控制标志启用。其执行线性插值，定义如下：

$$x = x_0 + \alpha(x_1 - x_0), \text{ for } 0 \leq \alpha < 1$$

其中为寄存器 `BASE0`，为寄存器 `BASE1`，且为由通道1的移位和掩码值最低有效8位构成的分数值。

混合模式与普通模式的区别如下：

- `PEEK0` 与 `POP0` 返回8位透明度值（即通道1移位和掩码值的最低8位），结果的第31位至第24位均为零。

- `PEEK1` 与 `POP1` 返回 `BASE0` 和 `BASE1` 之间的线性插值值。
- `PEEK2`, `POP2` 在加法中不包含车道1的结果（即它是 `BASE2` + 车道0的移位和掩码值）

当 `alpha` 值为0时，线性插值的结果等于 `BASE0`；当 `alpha` 值全为1时，结果等于 `BASE0 + 255/256 * (BASE1 - BASE0)`

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第68至87行

```

68 void simple_blend1() {
69     puts("简单混合1: ");
70
71     interp_config cfg = interp_default_config();
72     interp_config_set_blend(&cfg, true);
73     interp_set_config(interp0, 0, &cfg);
74
75     cfg = interp_default_config();
76     interp_set_config(interp0, 1, &cfg);
77
78     interp0->base[0] = 500;
79     interp0->base[1] = 1000;
80
81     for (int i = 0; i <= 6; i++) {
82         // 将fraction设置为介于0至255之间的值
83         interp0->accum[1] = 255 * i / 6;
84         // ≈ 500 + (1000 - 500) * i / 6;
85         printf("%d\n", (int) interp0->peek[1]);
86     }
87 }
```

应当打印（注意 `255/256` 导致结果为 `998`，而非 `1000`）：

```

500
582
666
748
832
914
998
```

`CTRL_LANE1_SIGNED` 控制 `BASE0` 与 `BASE1` 是否进行符号扩展以完成此插值（该符号扩展必须，因为插值生成了一个40位大小的中间乘积值）。`CTRL_LANE0_SIGNED` 继续正常控制 `PEEK2` 与 `POP2` 中 lane 0 中间结果的符号扩展。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第90至121行

```

90 void print_simple_blend2_results(bool is_signed) {
91     //通道1的有符号标志控制基数0/1是否被视为有符号或无符号
92     interp_config cfg = interp_default_config();
93     interp_config_set_signed(&cfg, is_signed);
94     interp_set_config(interp0, 1, &cfg);
95
96     for (int i = 0; i <= 6; i++) {
97         interp0->accum[1] = 255 * i / 6;
98         if (is_signed) {
99             printf("%d\n", (int) interp0->peek[1]);
100        } else {
101            printf("0x%08x\n", (uint) interp0->peek[1]);
102        }
103    }
104 }
105
106 void simple_blend2() {
107     puts("简单混合2:");
108
109     interp_config cfg = interp_default_config();
110     interp_config_set_blend(&cfg, true);
111     interp_set_config(interp0, 0, &cfg);
112
113     interp0->base[0] = (uint32_t) -1000;
114     interp0->base[1] = 1000;
115
116     puts("有符号:");
117     print_simple_blend2_results(true);
118
119     puts("unsigned:");
120     print_simple_blend2_results(false);
121 }
```

应打印：

```

signed:
-1000
-672
-336
-8
328
656
992
unsigned:
0xfffffc18
0xd5ffffd60
0xaafffeb0
0x80fffff8
0x56000148
0x2c000290
0x010003e0
```

最后，在混合模式下，使用 **BASE_1AND0** 寄存器通过单次32位写入向 **BASE0** 和 **BASE1** 各发送16位值时，这些16位值向完整32位值的符号扩展由 **CTRL_LANE1_SIGNED** 控制，适用于两个基寄存器。相比之下，非混合模式操作中，**CTRL_LANE0_SIGNED** 控制向 **BASE0** 的扩展，**CTRL_LANE1_SIGNED** 控制向 **BASE1** 的扩展。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第124至145行

```

124 void simple_blend3() {
125     puts("简单混合3:");
126
127     interp_config cfg = interp_default_config();
128     interp_config_set_blend(&cfg, true);
129     interp_set_config(interp0, 0, &cfg);
130
131     cfg = interp_default_config();
132     interp_set_config(interp0, 1, &cfg);
133
134     interp0->accum[1] = 128;
135     interp0->base01 = 0x30005000;
136     printf("0x%08x\n", (int) interp0->peek[1]);
137     interp0->base01 = 0xe000f000;
138     printf("0x%08x\n", (int) interp0->peek[1]);
139
140     interp_config_set_signed(&cfg, true);
141     interp_set_config(interp0, 1, &cfg);
142
143     interp0->base01 = 0xe000f000;
144     printf("0x%08x\n", (int) interp0->peek[1]);
145 }
```

应打印：

```

0x00004000
0x0000e800
0xfffffe800
```

2.3.1.6.3. 夹紧模式

夹紧模式适用于每个核心上的 `INTERP1`，且由 `CTRL_LANE0_CLAMP` 控制标志启用。在夹紧模式下，`PEEK0/POP0` 的结果为通道值（经过 `ACCU0` 的移位、掩码及符号扩展），该值被限制在 `BASE0` 与 `BASE1` 之间。换言之，若通道值大于 `BASE1`，则结果为 `BASE1`；若小于 `BASE0`，则结果为 `BASE0`；否则，值保持不变。不进行加法操作。上述比较的符号属性由 `CTRL_LANE0_SIGNED` 标志控制。

除此之外，插值器的行为与正常模式一致。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第193至211行

```

193 void clamp() {
194     puts("Clamp:");
195     interp_config cfg = interp_default_config();
196     interp_config_set_clamp(&cfg, true);
197     interp_config_set_shift(&cfg, 2);
198     // 根据符号位的新位置设置掩码。
199     interp_config_set_mask(&cfg, 0, 29);
200     // 以确保移位后的值正确进行符号扩展。
201     interp_config_set_signed(&cfg, true);
202     interp_set_config(interp1, 0, &cfg);
203
204     interp1->base[0] = 0;
205     interp1->base[1] = 255;
206
207     for (int i = -1024; i <= 1024; i += 256) {
```

```

208         interp1->accum[0] = i;
209         printf("%d\t%d\n", i, (int) interp1->peek[0]);
210     }
211 }
```

应打印：

```

-1024 0
-768 0
-512 0
-256 0
0 0
256 64
512 128
768 192
1024 255
```

2.3.1.6.4. 示例用例：线性插值

线性插值是结合混合模式及其他插值器功能的更完整示例：

在本示例中，`ACCUM0`用于跟踪待插值数值列表中的定点（整数/小数）位置。

通道0用于生成位置整数部分对应的值数组地址。位置的小数部分被移位以生成0至255范围内的混合值。混合操作在数组中相邻的两个值之间执行。

最后，通过对`ACCUM0_ADD_RAW`的单次写入操作更新小数位置。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第147至191行

```

147 void linear_interpolation() {
148     puts("线性插值:");
149     const int uv_fractional_bits = 12;
150
151     // 通道0
152     // 对XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (累加器0) 进行移位和掩码
153     // 至          0000 0000 000X XXXX XXXX XXXX XXXX XXX0 154
154     // 即整数部分乘以2 (用于uint16_t)
155     interp_config cfg = interp_default_config();
156     interp_config_set_shift(&cfg, uv_fractional_bits - 1);
157     interp_config_set_mask(&cfg, 1, 32 - uv_fractional_bits);
158     interp_config_set_blend(&cfg, true);
159     interp_set_config(interp0, 0, &cfg);
160
161     // 对通道1
162     // 将 XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (通过交叉输入累加为0)
163     // 移位至 0000 XXXX XXXX XXXX XXXX FFFF FFFF FFFF
164
165     cfg = interp_default_config();
166     interp_config_set_shift(&cfg, uv_fractional_bits - 8);
167     interp_config_set_signed(&cfg, true);
168     interp_config_set_cross_input(&cfg, true); // 有符号混合
169     interp_set_config(interp0, 1, &cfg);
170
171     int16_t samples[] = {0, 10, -20, -1000, 500};
172
173     // step 在我们的分数表示中为四分之一
```

```

174     uint step = (1 << uv_fractional_bits) / 4;
175
176     interp0->accum[0] = 0; // 初始 sample_offset;
177     interp0->base[2] = (uintptr_t) samples;
178     for (int i = 0; i < 16; i++) {
179         // result2 = samples + (lane0 原始结果)
180         // 即指向两个需混合样本中第一个样本的指针
181         int16_t *sample_pair = (int16_t *) interp0->peek[2];
182         interp0->base[0] = sample_pair[0];
183         interp0->base[1] = sample_pair[1];
184         uint32_t peek1 = interp0->peek[1];
185         uint32_t add_raw1 = interp0->add_raw[1];
186         printf("%d\t(%d%% 介于 %d 和 %d 之间)\n", (int) peek1,
187                100 * (add_raw1 & 0xff) / 0xff,
188                sample_pair[0], sample_pair[1]);
189         interp0->add_raw[0] = step;
190     }
191 }
```

应打印：

```

0      (0% 介于 0 和 10 之间)
2      (25% 介于 0 和 10 之间)
5      (50% 介于 0 和 10 之间)
7      (75% 介于 0 和 10 之间)
10     (0% 介于 10 和 -20 之间)
2      (25% 介于 10 和 -20 之间)
-5     (50% 介于 10 和 -20 之间)
-13    (75% 介于 10 和 -20 之间)
-20    (0% 介于 -20 和 -1000 之间)
-265   (25% 介于 -20 和 -1000 之间)
-510   (-20 与 -1000 之间的 50%)
-755   (-20 与 -1000 之间的 75%)
-1000  (-1000 与 500 之间的 0%)
-625   (-1000 与 500 之间的 25%)
-250   (-1000 与 500 之间的 50%)
125    (-1000 与 500 之间的 75%)
```

此方法用于 SDK 中的快速近似音频上采样。

2.3.1.6.5. 示例用例：简单仿射纹理映射

简单仿射纹理映射可通过对纹理坐标使用定点运算来实现，并在扫描线的每个像素处按固定步长更新坐标。纹理坐标的整数部分用于在纹理中形成地址，进而查找像素颜色。

通过使用两条通道、所有三个基值及`CTRL_LANEx_ADD_RAW`标志，可将原本较为昂贵的 CPU 操作简化为利用插值器执行的单周期迭代。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第 214 至 272 行

```

214 void texture_mapping_setup(uint8_t *texture, uint texture_width_bits, uint
215                             texture_height_bits,
216                             uint uv_fractional_bits) {
217     interp_config cfg = interp_default_config();
218     // 设置 add_raw 标志以在加法时使用原始（未移位且未屏蔽）的通道累加器值
219
220     // 将其加至通道基值以生成通道结果
```

```

219     interp_config_set_add_raw(&cfg, true);
220     interp_config_set_shift(&cfg, uv_fractional_bits);
221     interp_config_set_mask(&cfg, 0, texture_width_bits - 1);
222     interp_set_config(interp0, 0, &cfg);
223
224     interp_config_set_shift(&cfg, uv_fractional_bits - texture_width_bits);
225     interp_config_set_mask(&cfg, texture_width_bits, texture_width_bits +
        texture_height_bits - 1);
226     interp_set_config(interp0, 1, &cfg);
227
228     interp0->base[2] = (uintptr_t) texture;
229 }
230
231 void texture_mapped_span(uint8_t *output, uint32_t u, uint32_t v, uint32_t du, uint32_t dv,
   uint count) {
232     // u、v为具有uv_fractional_bits小数位的固定点纹理坐标
233     // du、dv为跨越该片段的纹理坐标步长，采用相同的固定点表示法。
234     interp0->accum[0] = u;
235     interp0->base[0] = du;
236     interp0->accum[1] = v;
237     interp0->base[1] = dv;
238     for (uint i = 0; i < count; i++) {
239         // 等同于
240         // uint32_t sm_result0 = (accum0 >> uv_fractional_bits) & (1 << (texture_width_bits -
        1));
241         // uint32_t sm_result1 = (accum1 >> uv_fractional_bits) & (1 << (texture_height_bits -
        1));
242         // uint8_t *address = texture + sm_result0 + (sm_result1 << texture_width_bits);
243         // output[i] = *address;
244         // accum0 = du + accum0;
245         // accum1 = dv + accum1;
246
247         // result2是当前像素的纹理地址;
248         // 弹出结果以便进行下一次迭代
249         output[i] = *(uint8_t *) interp0->pop[2];
250     }
251 }
252
253 void texture_mapping() {
254     puts("仿射纹理映射（带纹理重复）：");
255
256     uint8_t texture[] = {
257         0x00, 0x01, 0x02, 0x03,
258         0x10, 0x11, 0x12, 0x13,
259         0x20, 0x21, 0x22, 0x23,
260         0x30, 0x31, 0x32, 0x33,
261     };
262     // 4x4纹理
263     texture_mapping_setup(texture, 2, 2, 16);
264     uint8_t output[12];
265     uint32_t du = 65536 / 2; // 步长为1/2
266     uint32_t dv = 65536 / 3; // 步长为1/3
267     texture_mapped_span(output, 0, 0, du, dv, 12);
268
269     for (uint i = 0; i < 12; i++) {
270         printf("0x%02x\n", output[i]);
271     }
272 }

```

应打印：

```

0x00
0x00
0x01
0x01
0x12
0x12
0x12
0x13
0x23
0x20
0x20
0x20
0x31
0x31

```

2.3.1.7. 寄存器列表

SIO 寄存器地址起始于 **0xd0000000** (在 SDK 中定义为 SIO_BASE)。

表16. SIO
寄存器列表

偏移量	名称	说明
0x000	CPUID	处理器核心标识符
0x004	GPIO_IN	GPIO引脚输入值
0x008	GPIO_HI_IN	QSPI引脚输入值
0x010	GPIO_OUT	GPIO输出值
0x014	GPIO_OUT_SET	GPIO输出值设置
0x018	GPIO_OUT_CLR	GPIO输出值清除
0x01c	GPIO_OUT_XOR	GPIO输出值异或
0x020	GPIO_OE	GPIO输出使能
0x024	GPIO_OE_SET	GPIO输出使能设置
0x028	GPIO_OE_CLR	GPIO输出使能清除
0x02c	GPIO_OE_XOR	GPIO 输出使能 XOR
0x030	GPIO_HI_OUT	QSPI 输出值
0x034	GPIO_HI_OUT_SET	QSPI 输出值设置
0x038	GPIO_HI_OUT_CLR	QSPI 输出值清除
0x03c	GPIO_HI_OUT_XOR	QSPI 输出值 XOR
0x040	GPIO_HI_OE	QSPI 输出使能
0x044	GPIO_HI_OE_SET	QSPI 输出使能设置
0x048	GPIO_HI_OE_CLR	QSPI 输出使能清除
0x04c	GPIO_HI_OE_XOR	QSPI 输出使能 XOR
0x050	FIFO_ST	核间FIFO（邮箱）状态寄存器。
0x054	FIFO_WR	该核TX FIFO的写访问权限
0x058	FIFO_RD	该核RX FIFO的读访问权限
0x05c	SPINLOCK_ST	自旋锁状态
0x060	DIV_UDIVIDEND	除法器无符号被除数

偏移量	名称	说明
0x064	DIV_UDIVISOR	除法器无符号除数
0x068	DIV_SDIVIDEND	除法器有符号被除数
0x06c	DIV_SDIVISOR	除法器有符号除数
0x070	DIV_QUOTIENT	除法器结果商
0x074	DIV_REMAINDER	除法器结果余数
0x078	DIV_CSR	除法器控制与状态寄存器。
0x080	INTERP0_ACCUM0	累加器0的读写访问
0x084	INTERP0_ACCUM1	累加器1的读写访问
0x088	INTERP0_BASE0	BASE0寄存器的读写访问。
0x08c	INTERP0_BASE1	BASE1寄存器的读写访问。
0x090	INTERP0_BASE2	BASE2寄存器的读写访问。
0x094	INTERP0_POP_LANE0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作）。
0x098	INTERP0_POP_LANE1	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作）。
0x09c	INTERP0_POP_FULL	读取完整结果，同时将通道结果写入两个累加器（POP）。
0x0a0	INTERP0_PEEK_LANE0	读取LANE0结果，不改变任何内部状态（PEEK）。
0x0a4	INTERP0_PEEK_LANE1	读取LANE1结果，不改变任何内部状态（PEEK）。
0x0a8	INTERP0_PEEK_FULL	读取完整结果，不改变任何内部状态（PEEK）。
0x0ac	INTERP0_CTRL_LANE0	LANE 0 控制寄存器
0x0b0	INTERP0_CTRL_LANE1	LANE 1 控制寄存器
0x0b4	INTERP0_ACCUM0_ADD	写入的值会以原子方式累加至ACCUM0
0x0b8	INTERP0_ACCUM1_ADD	写入的值会以原子方式累加至ACCUM1
0x0bc	INTERP0_BASE_1AND0	写入时，低16位同时写入BASE0，高位写入BASE1。
0x0c0	INTERP1_ACCUM0	累加器0的读写访问
0x0c4	INTERP1_ACCUM1	累加器1的读写访问
0x0c8	INTERP1_BASE0	BASE0寄存器的读写访问。
0x0cc	INTERP1_BASE1	BASE1寄存器的读写访问。
0x0d0	INTERP1_BASE2	BASE2寄存器的读写访问。
0x0d4	INTERP1_POP_LANE0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作）。
0x0d8	INTERP1_POP_LANE1	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作）。
0x0dc	INTERP1_POP_FULL	读取完整结果，同时将通道结果写入两个累加器（POP）。
0x0e0	INTERP1_PEEK_LANE0	读取LANE0结果，不改变任何内部状态（PEEK）。

偏移量	名称	说明
0x0e4	INTERP1_PEEK_LANE1	读取LANE1结果，不改变任何内部状态（PEEK）。
0x0e8	INTERP1_PEEK_FULL	读取完整结果，不改变任何内部状态（PEEK）。
0x0ec	INTERP1_CTRL_LANE0	LANE 0 控制寄存器
0x0f0	INTERP1_CTRL_LANE1	LANE 1 控制寄存器
0x0f4	INTERP1_ACCUM0_ADD	写入的值会以原子方式累加至ACCUM0
0x0f8	INTERP1_ACCUM1_ADD	写入的值会以原子方式累加至ACCUM1
0x0fc	INTERP1_BASE_1AND0	写入时，低16位同时写入BASE0，高位写入BASE1。
0x100	SPINLOCK0	自旋锁寄存器 0
0x104	SPINLOCK1	自旋锁寄存器1
0x108	SPINLOCK2	自旋锁寄存器2
0x10c	SPINLOCK3	自旋锁寄存器3
0x110	SPINLOCK4	自旋锁寄存器4
0x114	SPINLOCK5	自旋锁寄存器5
0x118	SPINLOCK6	自旋锁寄存器6
0x11c	SPINLOCK7	自旋锁寄存器7
0x120	SPINLOCK8	自旋锁寄存器8
0x124	SPINLOCK9	自旋锁寄存器9
0x128	SPINLOCK10	自旋锁寄存器10
0x12c	SPINLOCK11	自旋锁寄存器11
0x130	SPINLOCK12	自旋锁寄存器12
0x134	SPINLOCK13	自旋锁寄存器13
0x138	SPINLOCK14	自旋锁寄存器14
0x13c	SPINLOCK15	自旋锁寄存器15
0x140	SPINLOCK16	自旋锁寄存器16
0x144	SPINLOCK17	自旋锁寄存器17
0x148	SPINLOCK18	自旋锁寄存器18
0x14c	SPINLOCK19	自旋锁寄存器19
0x150	SPINLOCK20	自旋锁寄存器20
0x154	SPINLOCK21	自旋锁寄存器21
0x158	SPINLOCK22	自旋锁寄存器22
0x15c	SPINLOCK23	自旋锁寄存器23
0x160	SPINLOCK24	自旋锁寄存器24
0x164	SPINLOCK25	自旋锁寄存器 25
0x168	SPINLOCK26	自旋锁寄存器 26
0x16c	SPINLOCK27	自旋锁寄存器 27

偏移量	名称	说明
0x170	SPINLOCK28	自旋锁寄存器 28
0x174	SPINLOCK29	自旋锁寄存器 29
0x178	SPINLOCK30	自旋锁寄存器 30
0x17c	SPINLOCK31	自旋锁寄存器 31

SIO: CPUID 寄存器

偏移: 0x000

描述

处理器核心标识符

表 17. CPUID
寄存器

位	描述	类型	复位值
31:0	从处理器核心 0 读取时, 值为 0; 从处理器核心 1 读取时, 值为 1。	只读	-

SIO: GPIO_IN 寄存器

偏移: 0x004

描述

GPIO引脚输入值

表 18. GPIO_IN
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	GPIO0 至 GPIO29 的输入值	只读	0x00000000

SIO: GPIO_HI_IN 寄存器

偏移: 0x008

描述

QSPI引脚输入值

表 19. GPIO_HI_IN
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	QSPI IO 端口 0 至 5 顺序的输入值: SCLK、SSn、SD0、SD1、SD2、SD3	只读	0x00

SIO: GPIO_OUT 寄存器

偏移: 0x010

描述

GPIO 输出值

表 20. GPIO_OUT
寄存器

位	描述	类型	复位值
31:30	保留。	-	-

位	描述	类型	复位值
29:0	设置GPIO0至29的输出电平（1/0 → 高/低）。 读回的值为最后写入的值，非引脚上的输入值。 如果核心0和核心1同时写入GPIO_OUT（或写入SET/CLR/XOR别名）， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00000000

SIO: GPIO_OUT_SET寄存器

偏移: 0x014

描述

GPIO输出值设置

表21。
GPIO_OUT_SE
T寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位设置操作，即 <code>GPIO_OUT = wdata</code>	WO	0x00000000

SIO: GPIO_OUT_CLR寄存器

偏移: 0x018

描述

GPIO输出值清除

表22。
GPIO_OUT_CL
R寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位清除操作，即 <code>GPIO_OUT &= ~wdata</code>	WO	0x00000000

SIO: GPIO_OUT_XOR寄存器

偏移: 0x01c

描述

GPIO输出值异或

表 23。
GPIO_OUT_XOR
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位异或操作，即 <code>GPIO_OUT ^= wdata</code>	WO	0x00000000

SIO: GPIO_OE 寄存器

偏移: 0x020

描述

GPIO 输出使能

表 24. GPIO_OE
寄存器

位	描述	类型	复位值
31:30	保留。	-	-

位	描述	类型	复位值
29:0	设置 GPIO0...29 的输出使能（1/0 → 输出/输入）。 读取操作返回最后写入的值。 如果核心0和核心1同时写入 GPIO_OE（或其SET/CLR/XOR别名寄存器）， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00000000

SIO: GPIO_OE_SET 寄存器

偏移: 0x024

描述

GPIO输出使能设置

表 25。
GPIO_OE_SET 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子位置位操作，即 $\text{GPIO_OE} = \text{wdata}$	WO	0x00000000

SIO: GPIO_OE_CLR 寄存器

偏移: 0x028

描述

GPIO输出使能清除

表 26。
GPIO_OE_CLR 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子位清除操作，即 $\text{GPIO_OE} \&= \sim \text{wdata}$	WO	0x00000000

SIO: GPIO_OE_XOR 寄存器

偏移: 0x02c

描述

GPIO 输出使能 XOR

表 27。
GPIO_OE_XOR
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子按位异或操作，即 $\text{GPIO_OE} \wedge= \text{wdata}$	WO	0x00000000

SIO: GPIO_HI_OUT 寄存器

偏移: 0x030

描述

QSPI 输出值

表 28。
GPIO_HI_OUT 寄存器

位	描述	类型	复位值
31:6	保留。	-	-

位	描述	类型	复位值
5:0	设置 QSPI IO0...5 的输出电平 (1/0 → 高/低)。 读回的值为最后写入的值，非引脚上的输入值。 若核心0与核心1同时写入 GPIO_HI_OUT (或其SET/CLR/XOR别名) ,则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00

SIO: GPIO_HI_OUT_SET 寄存器

偏移: 0x034

描述

QSPI 输出值设置

表 29.
GPIO_HI_OUT_SET
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子位设置操作，即 <code>GPIO_HI_OUT = wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_CLR 寄存器

偏移量: 0x038

描述

QSPI 输出值清除

表 30.
GPIO_HI_OUT_CLR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子位清除操作，即 <code>GPIO_HI_OUT &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_XOR 寄存器

偏移量: 0x03c

描述

QSPI 输出值异或

表 31.
GPIO_HI_OUT_XOR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子按位异或操作，即 <code>GPIO_HI_OUT ^= wdata</code>	WO	0x00

SIO: GPIO_HI_OE 寄存器

偏移量: 0x040

描述

QSPI 输出使能

表 32. GPIO_HI_OE
寄存器

位	描述	类型	复位值
31:6	保留。	-	-

位	描述	类型	复位值
5:0	设置 QSPI IO0...5 的输出使能 (1/0 → 输出/输入)。 读取操作返回最后写入的值。 当核心 0 和核心 1 同时写入 GPIO_HI_OE (或其 SET/CLR/XOR 别名) 时， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00

SIO: GPIO_HI_OE_SET 寄存器

偏移量: 0x044

描述

QSPI 输出使能设置

表 33.
GPIO_HI_OE_SET
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子位设置操作, 即 <code>GPIO_HI_OE = wdata</code>	WO	0x00

SIO: GPIO_HI_OE_CLR 寄存器

偏移: 0x048

描述

QSPI 输出使能清除

表 34.
GPIO_HI_OE_CLR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子位清除操作, 即 <code>GPIO_HI_OE &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OE_XOR 寄存器

偏移: 0x04c

描述

QSPI 输出使能 XOR

表 35.
GPIO_HI_OE_XOR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子按位异或操作, 即 <code>GPIO_HI_OE ^= wdata</code>	WO	0x00

SIO: FIFO_ST 寄存器

偏移: 0x050

描述

核间FIFO（邮箱）状态寄存器。

核心0到核心1方向有一个FIFO，核心1到核心0方向也有一个。两个FIFO均为32位宽，深度为8字。

核心0可访问 1→0 FIFO (RX) 的读端，以及 0→1 FIFO (TX) 的写端。

Core 1 可见 0→1 FIFO (RX) 的读取端，以及 1→0 FIFO (TX) 的写入端。

每个核心的 SIO IRQ 为其 FIFO_ST 寄存器中 VLD、WOF 和 ROE 字段的逻辑“或”运算结果。

表 36. FIFO_ST
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ROE : 粘滞标志, 表示 RX FIFO 在空时被读取。该读取操作被 FIFO 忽略。	WC	0x0
2	WOF : 粘滞标志, 表示 TX FIFO 在满时被写入。该写入操作被 FIFO 忽略。	WC	0x0
1	RDY : 当该核心的 TX FIFO 未满 (即 FIFO_WR 准备接收更多数据) 时, 值为 1。	只读	0x1
0	VLD : 当该核心的 RX FIFO 非空 (即 FIFO_RD 有效) 时, 值为 1。	只读	0x0

SIO: FIFO_WR 寄存器

偏移: 0x054

表 37. FIFO_WR
寄存器

位	描述	类型	复位值
31:0	该核TX FIFO的写访问权限	WF	0x00000000

SIO: FIFO_RD 寄存器

偏移: 0x058

表 38. FIFO_RD
寄存器

位	描述	类型	复位值
31:0	该核RX FIFO的读访问权限	RF	-

SIO: SPINLOCK_ST 寄存器

偏移: 0x05c

表 39.
SPINLOCK_ST
寄存器

位	描述	类型	复位值
31:0	自旋锁状态 包含所有32个自旋锁状态的位图 (1=已锁定)。 主要用于调试。	只读	0x00000000

SIO: DIV_UDIVIDEND 寄存器

偏移: 0x060

表 40.
DIV_UDIVIDEND
寄存器

位	描述	类型	复位值
31:0	除法器无符号被除数 写入除法器的DIVIDEND操作数, 即 p / q 中的 p 。 任何操作数的写入均启动新的计算。结果显示在QUOTIENT和REMAINDER寄存器中。 UDIVIDEND/SDIVIDEND为同一内部寄存器的别名。U别名启动无符号计算, S别名启动有符号计算。	读写	0x00000000

SIO: DIV_UDIVISOR 寄存器

偏移量: 0x064

表41。
DIV_UDIVISOR
寄存器

位	描述	类型	复位值
31:0	除法器无符号除数 写入除法器的除数操作数，即除式中p / q中的q。 任何操作数的写入均启动新的计算。结果显示在QUOTIENT和REMAINDER寄存器中。 UDIVISOR/SDIVISOR为同一内部寄存器的别名。U别名启动 S别名启动有符号计算。	读写	0x00000000

SIO: DIV_SDIVIDEND 寄存器

偏移量: 0x068

表42。
DIV_SDIVIDEND
寄存器

位	描述	类型	复位值
31:0	除法器有符号被除数 与UDIVIDEND相同，但启动有符号运算，而非无符号。	读写	0x00000000

SIO: DIV_SDIVISOR 寄存器

偏移量: 0x06C

表43。
DIV_SDIVISOR
寄存器

位	描述	类型	复位值
31:0	除法器有符号除数 与UDIVISOR相同，但启动有符号运算，而非无符号。	读写	0x00000000

SIO: DIV_QUOTIENT 寄存器

偏移量: 0x070

表44。
DIV_QUOTIENT
寄存器

位	描述	类型	复位值
31:0	除法器结果商 结果为DIVIDEND / DIVISOR (除法)。当CSR_READY为低电平时，内容未定义。 对于有符号计算，DIVIDEND与DIVISOR符号不同时，QUOTIENT为负。 该寄存器可直接写入，用于上下文保存/恢复目的。该操作将中断任何正在进行的计算，并设置CSR_READY及CSR_DIRTY标志。 读取QUOTIENT会清除CSR_DIRTY标志，因此若使用CSR_DIRTY，应按REMAINDER、QUOTIENT的顺序读取结果。	读写	0x00000000

SIO: DIV_REMAINDER 寄存器

偏移量: 0x074

表45。
DIV_REMAINDER
寄存器

位	描述	类型	复位值
31:0	除法器结果余数 结果为DIVIDEND % DIVISOR (取模)。当CSR_READY为低电平时，内容未定义。 对于有符号计算，仅当DIVIDEND为负时，REMAINDER才为负。 该寄存器可直接写入，用于上下文保存/恢复目的。该操作将中断任何正在进行的计算，并设置CSR_READY及CSR_DIRTY标志。	读写	0x00000000

SIO: DIV_CSR寄存器

偏移量: 0x078

描述

除法器控制与状态寄存器。

表46. DIV_CSR寄
存器

位	描述	类型	复位值
31:2	保留。	-	-
1	DIRTY : 当任何寄存器被写入时变为1，读取QUOTIENT时恢复为0。 软件可利用此标志提升保存/恢复操作的效率（非DIRTY时可跳过）。 若以此方式使用该标志，建议仅读取QUOTIENT，或先读取REMAINDER后读取QUOTIENT，以防上下文切换时数据丢失。	只读	0x0
0	READY : 计算进行中时读为0，完成后读为1。 写入操作数 (xDIVIDEND, xDIVISOR) 将立即启动新计算， 无论是否已有计算正在执行。 写入结果寄存器会立即中止任何当前计算， 并将READY与DIRTY标志设置为有效。	只读	0x1

SIO: INTERP0_ACCUM0寄存器

偏移量: 0x080

表47.
INTERP0_ACCUM
0寄存器

位	描述	类型	复位值
31:0	累加器0的读写访问	读写	0x00000000

SIO: INTERP0_ACCUM1寄存器

偏移: 0x084

表 48。
INTERP0_ACCUM1
寄存器

位	描述	类型	复位值
31:0	累加器1的读写访问	读写	0x00000000

SIO: INTERP0_BASE0 寄存器

偏移: 0x088

表 49。
INTERP0_BASE0
寄存器

位	描述	类型	复位值
31:0	BASE0寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_BASE1 寄存器

偏移: 0x08c

表 50。
INTERP0_BASE1
寄存器

位	描述	类型	复位值
31:0	BASE1寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_BASE2 寄存器

偏移: 0x090

表 51。
INTERP0_BASE2
寄存器

位	描述	类型	复位值
31:0	BASE2寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_POP_LANE0 寄存器

偏移: 0x094

表 52。
INTERP0_POP_LANE0
寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP0_POP_LANE1 寄存器

偏移: 0x098

表 53
INTERP0_POP_LANE1
寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP0_POP_FULL 寄存器

偏移: 0x09c

表 54
INTERP0_POP_FULL
寄存器

位	描述	类型	复位值
31:0	读取 FULL 结果，并同时将通道结果写入两个累加器 (POP)	只读	0x00000000

SIO: INTERP0_PEEK_LANE0 寄存器

偏移: 0x0a0

表 55
INTERP0_PEEK_LANE
0 寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP0_PEEK_LANE1 寄存器

偏移: 0x0a4

表 56
INTERPO_PEEK_LANE
1 寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERPO_PEEK_FULL 寄存器

偏移: 0x0a8

表57。
INTERPO_PEEK_FULL
寄存器

位	描述	类型	复位值
31:0	读取完整结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERPO_CTRL_LANE0 寄存器

偏移: 0x0ac

描述

LANE 0 控制寄存器

表58。
INTERPO_CTRL_LANE
0 寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25	OVERF: 如果 OVERF0 或 OVERF1 被置位，则该位被置位。	只读	0x0
24	OVERF1: 指示 ACCUM1 中任何被屏蔽的高有效位（MSBs）是否被置位。	只读	0x0
23	OVERF0: 指示 ACCUM0 中任何被屏蔽的高有效位（MSBs）是否被置位。	只读	0x0
22	保留。	-	-
21	BLEND: 仅存在于每个核心的 INTERPO 模块中。若启用 BLEND 模式： - LANE1 结果为 BASE0 与 BASE1 之间的线性插值，由 lane 1 的移位与掩码值的最低 8 位控制（介于 0 与 255/256 之间的分数值）。 - LANE0 结果不包含 BASE0（仅输出 lane 1 移位与掩码值的最低 8 位）。 - FULL 结果未加上车道 1 的位移+掩码值（BASE2 + 车道 0 位移+掩码） LANE1 SIGNED 标志用于控制插值的有符号或无符号属性。	读写	0x0
20:19	FORCE_MSB: 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW: 若设置为 1，则绕过 LANE0 结果的掩码与位移处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT: 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT: 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED: 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE0 之前，LANE0 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0

位	描述	类型	复位值
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP0_CTRL_LANE1 寄存器

偏移量: 0x0b0

说明

LANE 1 控制寄存器

表 59。
INTERP0_CTRL_LANE
1 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若值为 1，LANE1 结果将绕过掩码和移位处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT : 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED : 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE1 之前，LANE1 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP0_ACCUM0_ADD 寄存器

偏移量: 0x0b4

表 60。
INTERP0_ACCUM0_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至 ACCUM0 读取结果为 lane 0 的原始位移及掩码值（未加 BASE0）。	读写	0x000000

SIO: INTERP0_ACCUM1_ADD 寄存器

偏移: 0x0b8

表 61。
INTERP0_ACCUM1_A
DD寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM1 读取返回通道1的原始位移和值掩码（未加BASE1）。	读写	0x000000

SIO: INTERP0_BASE_1AND0 寄存器

偏移: 0x0bc

表 62。
INTERP0_BASE_1AN
D0寄存器

位	描述	类型	复位值
31:0	写入时，低16位同时写入BASE0，高位写入BASE1。 若该通道的SIGNED标志被设置，则每半部分将符号扩展至32位。	WO	0x00000000

SIO: INTERP1_ACCUM0 寄存器

偏移: 0x0c0

表 63。
INTERP1_ACCUM
0寄存器

位	描述	类型	复位值
31:0	累加器0的读写访问	读写	0x00000000

SIO: INTERP1_ACCUM1 寄存器

偏移: 0x0c4

表 64。
INTERP1_ACCUM
1寄存器

位	描述	类型	复位值
31:0	累加器1的读写访问	读写	0x00000000

SIO: INTERP1_BASE0 寄存器

偏移: 0x0c8

表 65。
INTERP1_BASE0
寄存器

位	描述	类型	复位值
31:0	BASE0寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_BASE1 寄存器

偏移: 0x0cc

表 66。
INTERP1_BASE1
寄存器

位	描述	类型	复位值
31:0	BASE1寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_BASE2 寄存器

偏移: 0x0d0

表 67。
INTERP1_BASE2
寄存器

位	描述	类型	复位值
31:0	BASE2寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_POP_LANE0 寄存器

偏移: 0x0d4

表 68。
INTERP1_POP_LANE0
寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP1_POP_LANE1 寄存器

偏移: 0x0d8

表 69。
INTERP1_POP_LANE1
寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP1_POP_FULL 寄存器

偏移: 0x0dc

表 70。
INTERP1_POP_FULL
寄存器

位	描述	类型	复位值
31:0	读取 FULL 结果，并同时将通道结果写入两个累加器 (POP)	只读	0x00000000

SIO: INTERP1_PEEK_LANE0 寄存器

偏移: 0x0e0

表 71。
INTERP1_PEEK_LANE
0 寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_PEEK_LANE1 寄存器

偏移: 0x0e4

表 72。
INTERP1_PEEK_LANE
1 寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_PEEK_FULL 寄存器

偏移: 0x0e8

表 73。
INTERP1_PEEK_FULL
寄存器

位	描述	类型	复位值
31:0	读取完整结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_CTRL_LANE0 寄存器

偏移: 0x0ec

说明

LANE 0 控制寄存器

表 74
INTERP1_CTRL_LANE
0 寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25	OVERF : 如果 OVERF0 或 OVERF1 被置位，则该位被置位。	只读	0x0
24	OVERF1 : 指示 ACCUM1 中任何被屏蔽的高有效位 (MSBs) 是否被置位。	只读	0x0
23	OVERF0 : 指示 ACCUM0 中任何被屏蔽的高有效位 (MSBs) 是否被置位。	只读	0x0

位	描述	类型	复位值
22	CLAMP : 仅存在于每个核心上的 INTERP1。如启用 CLAMP 模式: - LANE0 结果为移位并掩码处理后的 ACCUM0，受 BASE0 下限及 BASE1 上限约束。 - 这些比较的符号属性由 LANE0_CTRL_SIGNED 决定。	读写	0x0
21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若设置为 1，则绕过 LANE0 结果的掩码与位移处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT : 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED : 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE0 之前，LANE0 的 PEEK/POP 显示扩展为 32 位 由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP1_CTRL_LANE1 寄存器

偏移: 0x0f0

描述

LANE 1 控制寄存器

表 75
INTERP1_CTRL_LANE
1 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若值为 1，LANE1 结果将绕过掩码和移位处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0

位	描述	类型	复位值
16	CROSS_INPUT : 若设置为 1, 将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW, 亦会生效 (CROSS_INPUT 多路复用器位于位移与掩码绕过之前)。	读写	0x0
15	SIGNED : 若设置 SIGNED, 则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE1 之前, LANE1 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位 (包含) 若设置 MSB 小于 LSB, 可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位 (包含)	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP1_ACCUM0_ADD 寄存器

偏移: 0x0f4

表 76
INTERP1_ACCUM0_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM0 读取结果为 lane 0 的原始移位及掩码值 (未加 BASE0)。	读写	0x000000

SIO: INTERP1_ACCUM1_ADD 寄存器

偏移: 0x0f8

表 77。
INTERP1_ACCUM1_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM1 读取返回通道1的原始位移和值掩码 (未加BASE1)。	读写	0x000000

SIO: INTERP1_BASE_1AND0 寄存器

偏移: 0x0fc

表 78。
INTERP1_BASE_1AND0
D 寄存器

位	描述	类型	复位值
31:0	写入时, 低16位同时写入BASE0, 高位写入BASE1。 若该通道的SIGNED标志被设置, 则每半部分将符号扩展至32位。	WO	0x00000000

SIO: SPINLOCK0、SPINLOCK1、...、SPINLOCK30、SPINLOCK31 寄存器

偏移: 0x100、0x104、...、0x178、0x17c

表 79。SPINLOCK0、SPINLOCK1
、...、SPINLOCK30、SPINLOCK31
寄存器

位	描述	类型	复位值
31:0	<p>从自旋锁地址读取操作将：</p> <ul style="list-style-type: none"> - 若锁已被占用，返回0 - 否则返回非零值，并同时成功占用该锁 <p>写入（任何值）将释放该锁。</p> <p>若核心0与核心1同时尝试占用同一锁，核心0胜出。</p> <p>成功时返回的值为 $0x1 << \text{锁编号}$。</p>	读写	0x00000000

2.3.2. 中断

每个核心均配备了标准的ARM嵌套向量中断控制器（NVIC），具有32个中断输入。

每个NVIC所接收的中断相同，但GPIO中断例外：每个核心针对每个GPIO组各有一个GPIO中断。这些中断完全独立，例如核心0可被第0组的GPIO 0中断，而核心1可被同一组的GPIO 1中断。

在RP2040上，仅有低26个IRQ信号连接至NVIC，而IRQ 26至31固定为零（永不触发）。

通过向NVIC ISPR寄存器的第26至31位写入数据，核心仍可被强制进入相关中断处理程序。

表80. 中断

IRQ	中断源	IRQ	中断源	IRQ	中断源	IRQ	中断源	IRQ	中断源
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

① 注意

XIP_IRQ来源于组成XIP模块部分的SSI模块。该中断可用于代码从SRAM而非闪存运行的配置中。在此配置中，XIP块可用作普通SSI外设。

硬件支持嵌套中断：低优先级中断可被高优先级中断（或其他异常，例如 HardFault）抢占，低优先级中断将在高优先级异常处理完成后恢复执行。优先级顺序由以下因素决定：

- 首先，由 NVIC_IPR0-7 寄存器为每个中断配置的动态优先级级别决定。Cortex-M0+ 实现了8位优先级字段中最高的两位，因此共提供四个优先级级别，其中数值最低的级别（级别0）为最高优先级。
- 其次，对于动态优先级相同的中断，编号较低的 IRQ 具有更高优先级（依据上述表格中的 IRQ 编号）。

RP2040 中断表经过精心安排，提供合理的默认优先级顺序，但可通过 NVIC_IPR0 至 NVIC_IPR7 调整个别中断优先级，以满足特定使用需求。

26个系统IRQ信号被屏蔽（NMI屏蔽），然后通过逻辑或合并产生核心的NMI信号。每个核心的NMI屏蔽可通过Syscfg寄存器块中的PROC0_NMI_MASK和PROC1_NMI_MASK配置。这些寄存器的每个位对应一个系统中断，当系统中断被触发且对应核心的NMI屏蔽位被设置时，该核心的NMI信号即被触发。

⚠ 注意

若看门狗已启动，且核心1的NMI屏蔽中设置了某些位，则RESETS模块（包括Syscfg）应被纳入看门狗复位列表。否则，在看门狗事件发生后，当核心进入bootrom时，核心1的NMI可能会被触发。核心0在进入bootrom时接受NMI是安全的（处理程序会清除NMI屏蔽位）。

2.3.3. 事件信号

Cortex-M0+可使用 **WFE** 指令进入睡眠状态，直至发生“事件”（或中断）。它还可以使用 **SEV** 指令生成事件。在RP2040上，事件信号在两个处理器之间交叉连接，使得一个处理器发送的事件能够被另一个处理器接收。

ℹ 注意

事件标志具有“粘性”，因此若两个处理器同时发送事件（**SEV**），随后都进入休眠状态（**WFE**），两者会立即被唤醒，而不会陷入休眠。

处于 **WFE**（或 **WFI**）休眠状态时，处理器可以关闭其内部时钟门控，从而显著降低功耗。当两个处理器均处于休眠，且DMA处于非活动状态时，RP2040整体可以进入休眠状态，关闭未使用的基础设施时钟（如总线结构），并在任一处理器唤醒时自动复位。详见第2.11.2节。

2.3.4. 调试

2线制串行线调试（SWD）端口提供对硬件和软件调试功能的访问，包括：

- 将固件加载到SRAM或外部闪存中
- 处理器执行控制：运行/暂停、单步、设置断点及其他标准Arm调试功能
- 访问处理器架构状态
- 通过系统总线访问内存及内存映射IO

SWD总线通过两个专用引脚暴露，通电后即可使用。

ℹ 注意

我们建议最大SWD频率为24MHz。此设置高度依赖于您的具体配置。根据电缆的质量和长度，您可能需要以更低频率（1MHz）运行。

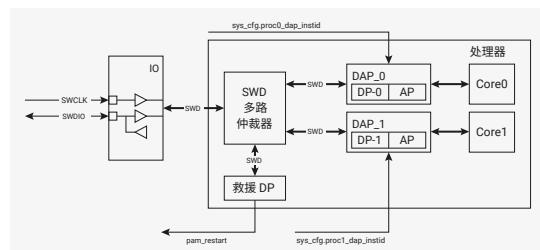
调试访问通过附着于共享多点SWD总线（SWD v2）的独立DAP（每核一个）实现。每个DAP仅在通过SWD **TARGETSEL**命令正确寻址后响应调试指令；所有其它DAP输出均处于高阻状态。

此外，提供一个Rescue DP（详见第2.3.4.2节），其连接至系统控制功能。各调试端口的默认地址如下：

- 核心 0: **0x01002927**
- 核心 1: **0x11002927**
- 救援 DP: **0xf1002927**

实例ID（上述ID的高4位）可通过sysconfig寄存器更改，这在多芯片应用中可能有所帮助。但请注意，ID=0xf保留用于内部救援DP（参见第2.3.4.2节）。

图 10. RP2040
调试



2.3.4.1. SWD引脚的软件控制

Core 0和Core 1的SWD引脚可通过syscfg寄存器（参见DBGFORCE）进行位操作控制。这意味着Core 1可以运行一个USB应用程序，从而实现对Core 0的调试，或其他类似功能。

2.3.4.2. Rescue DP

Rescue DP（调试端口）通过SWD总线提供，仅用于芯片死锁的特定情况，例如代码被烧录至闪存且系统时钟永久停止：在此情形下，普通调试器无法与处理器通信以恢复系统至正常工作状态，因此需采取更为严厉的措施。通过在Rescue DP的CTRL/STAT寄存器中设置CDBGPWRUPREQ位来触发救援操作。

此操作将引起芯片硬复位（功能上类似于上电复位），并在芯片级复位模块中设置标志，指示救援复位已发生。bootrom在初始化启动过程中（看门狗、闪存或USB启动之前）几乎立即检查此标志，确认后通过清除该位进行响应，随后停止处理器。系统因此处于安全状态，系统时钟持续运行，便于调试器重新连接核心并加载新的代码。

有关使用Rescue DP的实用示例，请参见《Hardware design with RP2040》一书。

2.4. Cortex-M0+

ARM文档

摘自Cortex-M0+技术参考手册，已获授权使用。

ARM Cortex-M0+处理器为极低门数、高能效处理器，设计用于需面积优化及低功耗的微控制器和深度嵌入式应用。

2.4.1. 功能特性

ARM Cortex-M0+处理器的特性与优势包括：

- 系统外设的紧密集成降低了面积及开发成本。
- Thumb指令集兼具高代码密度与32位性能。
- 支持单周期I/O访问。
- 对系统组件的电源控制进行了优化。
- 集成了低功耗睡眠模式。
- 快速代码执行使处理器能够以较低的时钟频率运行或延长休眠模式时间。

- 优化的代码取指以降低闪存和只读存储器的功耗。
- 硬件乘法器。
- 面向时间关键型应用的确定性高性能中断处理。
- 确定性的指令周期时序。
- 支持系统级调试认证。
- 串行线调试（Serial Wire Debug）减少调试所需的引脚数量。

2.4.1.1. 接口

处理器包含的用于外部访问的接口包括：

- 外部 AHB-Lite 接口至总线结构
- 调试访问端口（DAP）
- 单周期 I/O 端口至 SIO 外设

2.4.1.2. 配置

每个处理器配置有以下特性：

- 结构时钟门控（用于节能）
- 小端字节序总线访问
- 四个断点
- 调试支持（通过 2 线调试引脚 SWD/SWCLK）
- 32位指令取指（匹配32位数据总线）
- IOPORT（用于低延迟访问本地外设，详见SIO）
- 26个中断
- 8个MPU区域
- 所有寄存器上电复位
- 高速乘法器（MULS 32×32 单周期）
- SysTick定时器
- 向量表偏移寄存器（VTOR）
- 34条WIC（唤醒中断控制器）线路（32个IRQ和NMI，RXEV）
- DAP功能：支持停机事件
- DAP功能：SerialWire调试接口（协议2，支持多点连接）
- DAP功能：未实现微追踪缓冲区（MTB）

体系结构时钟门控允许处理器核心通过关闭部分核心时钟，支持SLEEP和DEEPSLEEP电源状态。请注意，不支持电源门控。

每个M0+核心都有独立的中断控制器，可根据需要单独屏蔽中断源。相同的中断信号被路由至两个M0+核心。

2.4.1.3. ARM 架构

该处理器实现了 ARMv6-M 架构概要。请参见《ARMv6-M 架构参考手册》，

更多详情请参考《ARM Cortex M0+ 技术参考手册》。

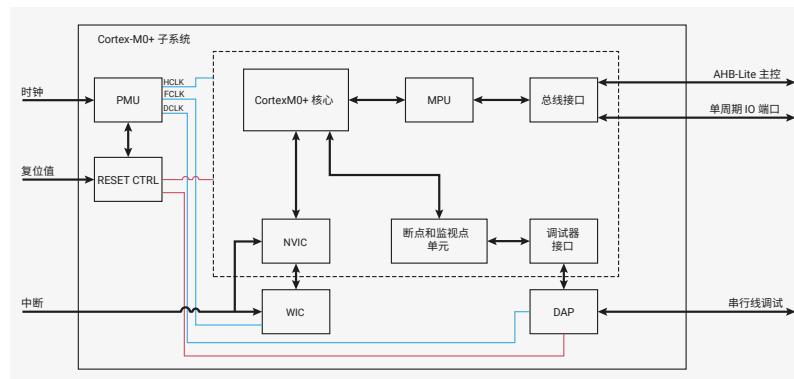
2.4.2. 功能描述

2.4.2.1. 概述

Cortex-M0+ 处理器是一款可配置的多级流水线 32 位 RISC 处理器。它具备 AMBA AHB-Lite 接口，并包含 NVIC 组件。同时还拥有硬件调试、单周期 I/O 接口及内存保护功能。该处理器能够执行 Thumb 代码，并与其他 Cortex-M 概要处理器兼容。

图 11 显示了处理器及其周边模块的功能模块。

图 11. Cortex M0+
功能模块
示意图



2.4.2.2. 特性

M0+ 的特性包括：

- ARMv6-M Thumb® 指令集。
- Thumb-2 技术。
- 符合 ARMv6-M 标准的 24 位 SysTick 计时器。
- 32 位硬件乘法器。这是标准的单周期乘法器。
- 具备确定性、固定延迟的中断处理能力。
- 支持可中止并可重启的多重加载/存储指令，以便实现快速中断处理。
- 符合 C 应用二进制接口的异常模型。这是 ARMv6-M 架构下符合 C 应用二进制接口（C-ABI）的异常模型，支持使用纯 C 函数作为中断处理程序。
- 采用 Wait For Interrupt (WFI)、Wait For Event (WFE) 指令，或中断返回时的睡眠退出功能，进入低功耗睡眠模式
-

2.4.2.3. NVIC 特性

嵌套向量中断控制器（NVIC）具备以下特性：

- 26 个外部中断输入，每个具备四个优先级等级。
- 专用的不可屏蔽中断 (NMI) 输入（可由任何标准中断源驱动）
- 支持电平触发和脉冲触发的中断线路。
- 唤醒中断控制器（WIC），实现超低功耗睡眠模式支持。
- 向量表可重定位。

注意

NVIC 支持硬件异常嵌套，例如当中断处理程序运行时，若有更高优先级中断请求到达，可中断当前处理程序。

详见第 2.4.5 节。

2.4.2.4. 调试功能

调试功能包括：

- 四个硬件断点。
- 两个监视点。
- 用于非侵入式代码分析的程序计数采样寄存器（PCSR）。
- 单步执行和向量捕获功能。
- 支持通过 BKPT 指令的无限制软件断点。
- 通过紧凑型总线矩阵对核心外设及零等待态系统从属设备进行非侵入式访问。调试器可在处理器运行时访问这些设备，包括内存。
- 处理器停止时，完全访问核心寄存器。
- 通过支持串行线调试连接的调试访问端口（DAP），实现符合 CoreSight 标准的调试访问。

2.4.2.4.1. 调试访问端口

处理器采用低门数调试访问端口（DAP）实现。低门数调试访问端口（DAP）提供串行线调试端口，并连接至处理器从属端口，提供完整的系统级调试访问。有关 DAP 的更多信息，请参见 ARM 调试接口 v5 架构规范的 ADI v5.1 版本。

2.4.2.5. MPU特性

内存保护单元（MPU）特性如下：

- 八个用户可配置的内存区域。
- 每个区域可禁用八个子区域。
- 支持执行禁止（XN）。
- 支持默认内存映射。

更多详细信息见第2.4.6节。

2.4.2.6. AHB-Lite接口

AHB-Lite接口上的传输始终标记为非连续。处理器访问与调试访问共享对外部AHB外设的外部接口。处理器访问优先于调试访问。

该总线可由任何厂商特定组件填充。

注意

指令仅通过AHB-Lite接口抓取。为优化性能，Cortex-M0+处理器预取执行指令之后的指令。为降低功耗，预取长度限制为最多32位。

2.4.2.7. 单周期I/O端口

处理器实现单周期I/O端口，提供对紧耦合外设（如通用输入输出(GPIO)）的高速访问。该端口可由处理器或调试器进行读写访问。不得从I/O端口执行代码。

2.4.2.8. 电源管理单元

每个处理器均配备独立的电源管理单元（PMU），通过关闭处理器核心部分的时钟实现节能。RP2040上不存在独立的电源域。

PMU由处理器时钟驱动，该时钟由芯片级时钟模块控制。PMU可控制处理器内部以下时钟域：

- 包含处理器调试资源及其余DAP的调试时钟。
- 包含NVIC的系统时钟。
- 包含核心及相关接口的处理器时钟。

控制限于时钟的启用与禁用。启用时，所有时钟域均以相同频率运行。

PMU还通过WIC接口，确保掉电和唤醒行为对软件透明，且符合时钟及睡眠需求。包括由SCR寄存器控制的SLEEP或DEEP SLEEP支持。

2.4.2.8.1. 电源管理

RP2040 ARM Cortex M0+采用ARMv6-M架构，支持在系统电源管理中使用Wait For Interrupt ([WFI](#)) 和 Wait For Event ([WFE](#)) 指令：

[WFI](#)提供硬件支持进入一个或多个睡眠状态的机制。硬件可以挂起执行，直到唤醒事件发生为止。

[WFE](#)提供软件挂起程序执行的机制，直到唤醒条件出现，对唤醒延迟的影响极小或无影响。[WFI](#)和[WFE](#)均为提示指令，可能对程序执行无任何影响。

通常，这些指令用于软件空闲循环，仅在发生中断或相关事件后恢复程序执行。

注意

使用[WFE](#)和[WFI](#)的代码须能处理因调试暂停或其他原因引起的虚假唤醒事件。

详情请参阅SDK及ARMv6-M指南。

2.4.2.8.2. 等待事件与发送事件

RP2040可通过使用Send-Event ([SEV](#)) 和 [WFE](#)提示指令支持基于软件的系统事件同步。软件可以：

- 使用[WFE](#)指令表示能够挂起进程或线程的执行，直至事件发生，
允许硬件进入低功耗状态。

- 依赖对软件透明且具备低延迟唤醒能力的机制。

WFE机制依赖硬件与软件协同实现节能。例如，暂停处理器执行，直到设备或另一处理器设置标志：

- 硬件提供进入 WFE 低功耗状态的机制。
- 软件进入轮询循环以监测标志状态：
- 若标志未被设置，轮询处理器将在轮询循环中发出 WFE 指令。
- 当标志被设置时，会产生事件（硬件中断或来自另一处理器的 Send-Event 指令）。

WFE 唤醒事件

以下事件是 WFE 唤醒事件：

- 在另一处理器上执行 SEV 指令
- 若系统控制寄存器中的SEVONPEND位设置为1，则任何进入挂起状态的异常。
- 优先级高于当前活动异常的异步异常。
- 启用调试时发生的调试事件。

事件寄存器

事件寄存器为单比特寄存器。设置时，事件寄存器表示自上次清除以来发生了可能阻止处理器因执行 WFE 指令而挂起的事件。事件寄存器适用以下条件：

- 复位会清除事件寄存器。
- 任何 WFE 唤醒事件或异常返回指令的执行均会设置事件寄存器。
- 执行 WFE 指令会清除事件寄存器。
- 软件无法直接读写事件寄存器的值。

发送事件指令

发送事件（SEV）指令会向另一处理器发送事件信号。Send-Event 指令产生唤醒事件。

Wait For Event 指令

WFE 指令的操作依赖于事件寄存器的状态：

- 若事件寄存器被置位，该指令将清除寄存器并立即返回。
- 若事件寄存器未被置位，处理器可挂起执行并进入低功耗状态。处理器可持续处于该状态，直至检测到 WFE 唤醒事件或复位信号。当处理器检测到 WFE 唤醒事件时，WFE 指令完成执行。

WFE 唤醒事件可发生于 WFE 指令发出之前。应用 WFE 机制的软件必须能够容忍虚假唤醒事件，包括多次唤醒。

2.4.2.8.3. 等待中断

RP2040 通过提示指令 WFI 支持等待中断功能。

处理器发出 WFI 指令后，可挂起执行并进入低功耗状态。处理器可持续处于该状态，直至检测到以下任一 WFI 唤醒事件：

- 复位。
- 一个异步异常，其优先级若PRIMASK.PM设置为0，将抢占任何当前处于活动状态的异常。

注意

若 **PRIMASK.PM**设置为1，且异步异常的组优先级高于任何活动异常，则会导致 **WFI**指令退出。若异常的组优先级小于或等于执行组优先级，该异常将被忽略。

- 如果启用调试，则触发调试事件。
- 一个 **WFI**唤醒事件。

当硬件检测到**WFI**唤醒事件时，**WFI**指令即完成。

处理器仅在发出 **WFI** 指令后识别**WFI**唤醒事件。

2.4.2.8.4. 唤醒中断控制器

唤醒中断控制器（WIC）用于根据 **SCR**寄存器的控制，将处理器从DEEPSLEEP状态唤醒。在DEEPSLEEP状态下，处理器核心及NVIC的时钟处于停止状态。从深度睡眠（DEEPSLEEP）状态唤醒可能需要几个周期。

WIC 接收来自另一处理器的接收事件信号、32条中断线及非屏蔽中断（NMI）作为输入。

为了进一步节省功耗，RP2040 支持第2.11节定义的系统级节能模式，内附代码示例。

2.4.2.9. 复位控制

Cortex M0+ 复位控制模块管理以下复位：

- 调试复位
- M0+ 核心复位
- PMU 复位

上电后，两个处理器均解除复位（详见第2.13.2节），释放调试、M0+ 核心及 PMU 的复位信号。

运行时，复位可由调试器、NVIC（通过**AIRCR.SYSRESETREQ**）或 RP2040 上电状态机控制器触发（详见第2.13节）。NVIC 仅复位 Cortex-M0+ 处理器核心（不包括调试及 PMU），而上电状态机控制器则可复位处理器子系统，断言该子系统所有复位信号（调试、M0+ 核心及 PMU）。

2.4.3. 程序员模型

2.4.3.1 关于程序员模型

ARMv6-M 架构参考手册提供了程序员模型的完整描述。本章概述了Cortex-M0+程序员模型，说明了实现定义的选项。其中还包含该处理器所使用的ARMv6-M Thumb指令及其周期数。更多详细内容请参见后续章节。

- 第2.4.4节总结了程序员模型的系统控制功能。
- 第2.4.5节总结了程序员模型的NVIC功能。
- 第2.3.4节总结了程序员模型的调试功能。

2.4.3.2 操作模式与执行

有关操作模式与执行的详情，请参阅ARMv6-M架构参考手册。

2.4.3.3 指令集概要

该处理器实现了ARMv6-M Thumb指令集，包括若干采用Thumb-2技术的32位指令。ARMv6-M 指令集包括：

- ARMv7-M 中除 CBZ、CBNZ 和 IT 以外的所有 16 位 Thumb 指令。
- 32 位 Thumb 指令：BL、DMB、DSB、ISB、MRS 和 MSR。

表 81 显示了 Cortex-M0+ 指令及其周期数。周期数基于零等待状态的系统。

表 81. Cortex-M0+
指令摘要

操作	描述	汇编指令	周期数
移动	8 位立即数	MOVS Rd, #<imm>	1
	低寄存器至低寄存器	MOVS Rd, Rm	1
	任意寄存器至任意寄存器	MOV Rd, Rm	1
	任意寄存器至 PC	MOV PC, Rm	2
加法	3位立即数	ADDS Rd, Rn, #<imm>	1
	所有寄存器 Lo	ADDS Rd, Rn, Rm	1
	任意寄存器至任意寄存器	ADD Rd, Rd, Rm	1
	任意寄存器至 PC	ADD PC, PC, Rm	2
减法	8 位立即数	ADDS Rd, Rd, #<imm>	1
	含进位	ADCS Rd, Rd, Rm	1
	立即数加至 SP	ADD SP, SP, #<imm>	1
	由 SP 生成地址	ADD Rd, SP, #<imm>	1
乘法	由 PC 生成地址	ADR Rd, <label>	1
	Lo 和 Lo	SUBS Rd, Rn, Rm	1
	3位立即数	SUBS Rd, Rn, #<imm>	1
	8 位立即数	SUBS Rd, Rd, #<imm>	1
比较	含进位	SBCS Rd, Rd, Rm	1
	来自 SP 的立即数	SUB SP, SP, #<imm>	1
	取负	RSBS Rd, Rn, #0	1
	乘法	MULS Rd, Rm, Rd	1
逻辑	比较	CMP Rn, Rm	1
	负值	CMN Rn, Rm	1
	立即数	CMP Rn, #<imm>	1
逻辑	与	ANDS Rd, Rd, Rm	1
	异或	EORS Rd, Rd, Rm	1
	或	ORRS Rd, Rd, Rm	1

操作	描述	汇编指令	周期数
	位清零	BICS Rd, Rd, Rm	1
	取反并移动	MVNS Rd, Rm	1
	与测试	TST Rn, Rm	1
移位操作	立即数逻辑左移	LSLS Rd, Rm, #<shift>	1
	寄存器逻辑左移	LSLS Rd, Rd, Rs	1
	立即数逻辑右移	LSRS Rd, Rm, #<shift>	1
	寄存器逻辑右移	LSRS Rd, Rd, Rs	1
	算术右移	ASRS Rd, Rm, #<shift>	1
	寄存器算术右移	ASRS Rd, Rd, Rs	1
旋转	寄存器右旋转	RORS Rd, Rd, Rs	1
加载	字, 立即偏移	LDR Rd, [Rn, #<imm>]	2 或 1 ^a
	半字, 立即偏移	LDRH Rd, [Rn, #<imm>]	2 或 1 ^a
	字节, 立即偏移	LDRB Rd, [Rn, #<imm>]	2 或 1 ^a
	字, 寄存器偏移	LDR Rd, [Rn, Rm]	2 或 1 ^a
	半字, 寄存器偏移	LDRH Rd, [Rn, Rm]	2 或 1 ^a
	带符号半字, 寄存器偏移	LDRSH Rd, [Rn, Rm]	2 或 1 ^a
	字节, 寄存器偏移	LDRB Rd, [Rn, Rm]	2 或 1 ^a
	带符号字节, 寄存器偏移	LDRSB Rd, [Rn, Rm]	2 或 1 ^a
	PC 相对	LDR Rd, <label>	2 或 1 ^a
	SP 相对	LDR Rd, [SP, #<imm>]	2 或 1 ^a
	多重载入, 排除基址	LDM Rn!, {<loreglist>}	1+N ^b
	多重载入, 包括基址	LDM Rn, {<loreglist>}	1+N ^b
存储	字, 立即偏移	STR Rd, [Rn, #<imm>]	2 或 1 ^a
	半字, 立即偏移	STRH Rd, [Rn, #<imm>]	2 或 1 ^a
	字节, 立即偏移	STRB Rd, [Rn, #<imm>]	2 或 1 ^a
	字, 寄存器偏移	STR Rd, [Rn, Rm]	2 或 1 ^a
	半字, 寄存器偏移	STRH Rd, [Rn, Rm]	2 或 1 ^a
	字节, 寄存器偏移	STRB Rd, [Rn, Rm]	2 或 1 ^a
	SP 相对	STR Rd, [SP, #<imm>]	2 或 1 ^a
	多重	STM Rn!, {<loreglist>}	1+N ^b
压入	压入	PUSH {<loreglist>}	1+N ^b
	带链接寄存器的压入	PUSH {<loreglist>, LR}	1+N ^c
弹出	弹出	POP {<loreglist>}	1+N ^b
	弹出并返回	POP {<loreglist>, PC}	3+N ^c
分支	条件分支	B<cc> <label>	1 或 2 ^d
	无条件分支	B <label>	2

操作	描述	汇编指令	周期数
	带链接	BL <label>	3
	带交换	BX Rm	2
	带链接和交换	BLX Rm	2
扩展	有符号半字扩展至字	SXTB Rd, Rm	1
	有符号字节扩展至字	SXTB Rd, Rm	1
	无符号半字	UXTH Rd, Rm	1
	无符号字节	UXTB Rd, Rm	1
字节反转	字中的字节	REV Rd, Rm	1
	两个半字中的字节	REV16 Rd, Rm	1
	有符号下半字	REVSH Rd, Rm	1
状态	更改主管调用 (SVC)	SVC #<imm>	- e
	禁用中断	CPSID i	1
	使能中断	CPSIE i	1
	读取特殊寄存器	MRS Rd, <specreg>	3
	写入特殊寄存器	MSR <specreg>, Rn	3
	断点	BKPT #<imm>	- e
提示	发送事件	SEV	1
	等待事件	WFE	2 ^f
	等待中断	WFI	2 ^f
	让出	YIELD	1 ^f
	无操作	NOP	1
屏障	指令同步	ISB	3
	数据存储器	DMB	3
	数据同步	DSB	3

表格注释

^a 当连接至AHB接口或SCS时为2，连接至单周期I/O端口时为1。

^b N为列表中元素的数量。

^c N为包含PC或LR的列表中元素数量。

^d 跳转时为2，不跳转时为1。

^e 周期数取决于处理器及调试配置。

^f 不计入等待中断或事件的时间。

^g 作为NOP指令执行。

有关ARMv6-M Thumb指令的详细信息，请参阅《ARMv6-M架构参考手册》。

2.4.3.4. 内存模型

该处理器包含一个总线矩阵，用于仲裁处理器核心和调试访问端口（DAP）对外部存储系统以及内部NVIC和调试组件的内存访问。

优先权始终赋予处理器，以确保所有调试访问尽可能不具侵入性。对于零

等待状态系统，所有对系统内存、NVIC及调试资源的调试访问在典型代码执行时均完全不具侵入性。

系统内存映射符合ARMv6-M架构规范，且对调试器与处理器的访问均通用。事务路由如下：

- 所有低于 `0xd0000000` 或高于 `0xefffffff` 的访问，均作为 AHB-Lite 事务出现在处理器的 AHB-Lite 主端口。
- 位于范围 `0xd0000000` 至 `0xdfffffff` 的访问由 SIO 处理。
- 访问范围 `0xe0000000` 至 `0xefffffff` 在处理器内部处理，不会出现在处理器的 AHB-Lite 主端口。

处理器仅支持范围 `0xd` 内的字（word）大小访问 `00000000-0xefffffff`。

表82显示了默认内存映射中每个区域的代码、数据和设备适用性。此内存映射在禁用 MPU 时使用。除针对 Cortex-M0+ NVIC 及调试组件的区域外，所有区域的属性和权限均可通过已实现的 MPU 进行修改。

表82. M0+默认内存映射使用情况

地址范围	代码	数据	设备
<code>0xf0000000 - 0xffffffff</code>	否	否	是
<code>0xe0000000 - 0xefffffff</code>	否	否	否 ^a
<code>0xa0000000 - 0xdfffffff</code>	否	否	是
<code>0x60000000 - 0x9fffffff</code>	是	是	否
<code>0x40000000 - 0x5fffffff</code>	否	否	是
<code>0x20000000 - 0x3fffffff</code>	是	是	否
<code>0x00000000 - 0x1fffffff</code>	是	是	否

^a. 为 Cortex-M0+ NVIC 及调试组件预留的空间。

注意

未标记为可执行代码的区域视为禁止执行（eXecute-Never, XN）；若代码尝试从该位置执行，将引发HardFault异常。

有关内存模型的详细信息，请参阅ARMv6-M架构参考手册。

2.4.3.5 处理器核寄存器摘要

表83列示了处理器核寄存器集合的摘要。每个寄存器宽度均为32位。

表83. M0+处理器核寄存器集合摘要

名称	描述
R0-R12	R0-R12为用于数据操作的通用寄存器。
MSP/PSP (R13)	堆栈指针（SP）是寄存器R13。在线程模式下，CONTROL寄存器指示使用的堆栈指针，即主堆栈指针（MSP）或进程堆栈指针（PSP）。
LR (R14)	链接寄存器（LR）是寄存器R14。它存储子程序、函数调用及异常的返回地址。
PC (R15)	程序计数器（PC）是寄存器R15，包含当前程序的地址。

名称	描述
PSR	程序状态寄存器（PSR）包含： <ul style="list-style-type: none">• 应用程序状态寄存器（APSR）• 中断程序状态寄存器（IPSR）• 执行程序状态寄存器（EPSR） 这些寄存器分别提供了对PSR的不同视图。
PRIMASK	PRIMASK寄存器禁止所有具有可配置优先级的异常被激活。
CONTROL	CONTROL寄存器在处理器处于线程模式时，控制所使用的堆栈及代码权限级别。

注意

有关处理器核心寄存器的地址、访问类型及复位值等信息，请参阅ARMv6-M架构参考手册。

2.4.3.6. 异常

本节介绍处理器的异常模型。

2.4.3.6.1. 异常处理

处理器实现了高级异常和中断处理，如《ARMv6-M 架构参考手册》中所述。为最小化中断延迟，处理器在响应任何待处理中断时，会放弃正在执行的多重加载（load-multiple）或多重存储（store-multiple）指令。从中断处理程序返回时，处理器将从头重新执行该多重加载或多重存储指令。

这意味着软件在访问只读敏感或对重复写入敏感的内存区域中的设备时，不得使用多重加载或多重存储指令。在任何可能导致不一致结果或不良副作用的重复读取或写入情况下，软件均不得使用此类指令。

处理器设计保证NVIC检测中断信号并使处理器取指向相关中断处理程序首条指令所需的周期数为固定值。如此一来，最高优先级中断将实现无抖动响应。这将取决于中断处理程序的位置以及是否有其他更高优先级的主控访问该内存。提供了SRAM4和SRAM5，可分配给各处理器的中断处理程序，因此无抖动。

为了减少中断延迟和抖动，Cortex-M0+处理器实现了ARMv6-M架构定义的中断后到达和中断尾部链接机制。在零等待状态系统中且未采用抖动抑制的情况下，最高优先级激活中断的最坏中断延迟为15个时钟周期。

处理器异常模型除架构规定的行为外，还具有以下实现定义行为：

- 从HardFault到NMI的堆栈异常将在NMI优先级下锁死。
- 从NMI到HardFault的弹栈异常将在HardFault优先级下锁死。

2.4.4. 系统控制

2.4.4.1 系统控制寄存器概述

表84列出了系统控制寄存器。每个寄存器宽度均为32位。

表84。M0+ 系统控制寄存器

名称	描述
SYST_CSR	SysTick控制与状态寄存器
SYST_RVR	SysTick 重装载值寄存器
SYST_CVR	SysTick 当前值寄存器
SYST_CALIB	SysTick 校准值寄存器
CPUID	参见 CPUID 寄存器
ICSR	中断控制和状态寄存器
AIRCR	应用程序中断及复位控制寄存器
CCR	配置与控制寄存器
SHPR2	系统处理优先级寄存器
SHPR3	系统处理优先级寄存器
SHCSR	系统处理控制和状态寄存器
VTOR	向量表偏移寄存器
ACTLR	辅助控制寄存器

注意

- 所有系统控制寄存器仅能通过字传输进行访问。任何尝试读取或写入半字或字节的操作均属不可预测。
- 有关系统控制寄存器及其地址、访问类型和复位值的详细信息，请参阅《寄存器列表》或《ARMv6-M体系结构参考手册》。

2.4.4.1.1. CPUID寄存器

CPUID寄存器包含特定于处理器的部件编号、版本和实现信息。

！重要

该标准内部Arm寄存器包含有关处理器类型的信息。其不应与CPUID（第2.3.1.1节）混淆，后者为RP2040 S10寄存器，在核心0上读取值为0，在核心1上读取值为1。

2.4.5. NVIC

2.4.5.1. 关于NVIC

外部中断信号连接至嵌套向量中断控制器（NVIC），NVIC负责中断的优先级排序。软件可以设置各中断的优先级。NVIC与Cortex-M0+处理器核心紧密耦合，提供低延迟的中断处理及对后发中断的高效处理能力。

注意

“嵌套”指中断能够被优先级更高的中断打断。“向量化”指硬件将每个中断分派至由向量表指定的独立处理程序例程。有关嵌套和向量行为的详细描述，见 ARMv6-M 架构参考手册。

所有 NVIC 寄存器仅可通过字（word）传输访问。任何单独读取或写入半字（halfword）或字节（byte）均属于不可预测操作。

NVIC 寄存器始终采用小端序格式。

处理器异常处理详见“异常”章节。

2.4.5.1.1. SysTick 计时器

24 位 SysTick 系统计时器，扩展处理器及 NVIC 功能，提供如下功能：

- 24 位系统计时器（SysTick）。
- 额外的可配置优先级 SysTick 中断。

SysTick 计时器使用 $1\mu s$ 脉冲作为时钟使能信号。此信号作为 timer_tick 在看门狗模块中生成。SysTick 定时的准确性依赖于该 timer_tick 的准确性。SysTick 定时器亦可由系统时钟驱动（参见 SYST_CALIB）。

欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

2.4.5.1.2. 低功耗模式

该实现包含 WIC，允许处理器及 NVIC 进入极低功耗休眠模式，由 WIC 负责识别与优先处理中断。

处理器完整支持 Wait For Interrupt (WFI)、Wait For Event (WFE) 及 Send Event (SEV) 指令。此外，处理器亦支持 SLEEP/NEXIT 功能，使处理器内核在异常处理程序返回线程模式时进入睡眠状态。欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

2.4.5.2. NVIC 寄存器概述

表 85 显示了 NVIC 寄存器。每个寄存器宽度均为 32 位。

表 85. M0+ NVIC 寄存器

名称	描述
NVIC_ISER	中断使能置位寄存器。
NVIC_ICER	中断使能清除寄存器。
NVIC_ISPR	中断挂起置位寄存器。
NVIC_ICPR	中断挂起清除寄存器。
NVIC_IPR0 - NVIC_IPR7	中断优先级寄存器。

注意

有关 NVIC 寄存器及其地址、访问类型及复位值的详细信息，请参阅寄存器列表或 ARMv6-M 架构参考手册。

2.4.6. MPU

2.4.6.1 关于MPU

MPU是实现内存保护的组件，使处理器能够支持ARMv6受保护内存系统架构模型。MPU全面支持以下功能：

- 八个统一的保护区域。
- 支持重叠的保护区域，区域优先级递增：
 - 7 = 最高优先级。
 - 0 = 最低优先级。
- 访问权限控制。
- 向系统导出内存属性。

MPU不匹配和权限违规将触发HardFault异常处理程序。欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

您可以使用MPU来实现：

- 执行特权规则。
- 分离进程。
- 管理内存属性。

2.4.6.2. MPU寄存器概述

表86显示了MPU寄存器。每个寄存器宽度均为32位。

表86. M0+ MPU
寄存器

名称	描述
MPU_TYPE	MPU类型寄存器。
MPU_CTRL	MPU控制寄存器。
MPU_RNR	MPU区域编号寄存器。
MPU_RBAR	MPU区域基地址寄存器。
MPU_RASR	MPU区域属性及大小寄存器。

注意

- 有关MPU寄存器及其地址、访问类型和复位值的详细信息，请参见ARMv6-M架构参考手册。
- MPU支持256字节至4Gb的区域大小，每个区域包含8个子区域。

2.4.7. 调试

基本调试功能包括处理器停止、单步执行、处理器核心寄存器访问、复位和硬故障向量捕获、无限制软件断点及完整系统内存访问。请参见ARMv6-M架构参考手册。

本设备的调试功能包括：

- 支持4个硬件断点的断点单元。

- 支持2个观察点的观察点单元。

2.4.8. 寄存器列表

ARM Cortex-M0+ 寄存器起始基址为 `0xe0000000`（在 SDK 中定义为 `PPB_BASE`）。

表 87。
M0PLUS 寄存器列表

偏移量	名称	说明
0xe010	<code>SYST_CSR</code>	SysTick控制与状态寄存器
0xe014	<code>SYST_RVR</code>	SysTick 重装载值寄存器
0xe018	<code>SYST_CVR</code>	SysTick 当前值寄存器
0xe01c	<code>SYST_CALIB</code>	SysTick 校准值寄存器
0xe100	<code>NVIC_ISER</code>	中断使能设置寄存器
0xe180	<code>NVIC_ICER</code>	中断使能清除寄存器
0xe200	<code>NVIC_ISPR</code>	中断挂起设置寄存器
0xe280	<code>NVIC_ICPR</code>	中断挂起清除寄存器
0xe400	<code>NVIC_IPR0</code>	中断优先级寄存器 0
0xe404	<code>NVIC_IPR1</code>	中断优先级寄存器1
0xe408	<code>NVIC_IPR2</code>	中断优先级寄存器2
0xe40c	<code>NVIC_IPR3</code>	中断优先级寄存器3
0xe410	<code>NVIC_IPR4</code>	中断优先级寄存器4
0xe414	<code>NVIC_IPR5</code>	中断优先级寄存器5
0xe418	<code>NVIC_IPR6</code>	中断优先级寄存器6
0xe41c	<code>NVIC_IPR7</code>	中断优先级寄存器7
0xed00	<code>CPUID</code>	CPUID基准寄存器
0xed04	<code>ICSR</code>	中断控制与状态寄存器
0xed08	<code>VTOR</code>	向量表偏移寄存器
0xed0c	<code>AIRCR</code>	应用程序中断及复位控制寄存器
0xed10	<code>SCR</code>	系统控制寄存器
0xed14	<code>CCR</code>	配置与控制寄存器
0xed1c	<code>SHPR2</code>	系统处理器优先级寄存器2
0xed20	<code>SHPR3</code>	系统处理器优先级寄存器3
0xed24	<code>SHCSR</code>	系统处理控制和状态寄存器
0xed90	<code>MPU_TYPE</code>	MPU类型寄存器
0xed94	<code>MPU_CTRL</code>	MPU控制寄存器
0xed98	<code>MPU_RNR</code>	MPU区域编号寄存器
0xed9c	<code>MPU_RBAR</code>	MPU 区域基址寄存器
0xedaa	<code>MPU_RASR</code>	MPU 区域属性与大小寄存器

M0PLUS: SYST_CSR 寄存器

偏移: 0xe010

描述

使用 SysTick 控制与状态寄存器启用 SysTick 功能。

表 88. SYST_CSR
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	COUNTFLAG ：若计时器自上次读取后计数到 0，则返回 1。 由应用程序或调试器读取时清除该标志。	只读	0x0
15:3	保留。	-	-
2	CLKSOURCE ：SysTick 时钟源。若 SYST_CALIB 报告 NOREF，则始终读取为 1。 选择 SysTick 计时器时钟源： 0 = 外部参考时钟。 1 = 处理器时钟。	读写	0x0
1	TICKINT ：启用 SysTick 异常请求： 0 = 计数到零时不发生 SysTick 异常请求。 1 = 计数到零时发生 SysTick 异常请求。	读写	0x0
0	ENABLE ：启用 SysTick 计数器： 0 = 计数器已禁用。 1 = 计数器已启用。	读写	0x0

M0PLUS: SYST_RVR 寄存器

偏移量: 0xe014

说明

使用 SysTick 重载值寄存器指定计数器计数至 0 时加载到当前值寄存器的起始值。该值可为介于 0 至 0xFFFFFFFF 之间的任意值。起始值可设为 0，但无效，因为 SysTick 中断及 COUNTFLAG 仅在从 1 计数到 0 时激活。该寄存器的复位值未知。

欲生成周期为 N 个处理器时钟周期的多次定时器，请将 RELOAD 设为 N-1。例如，若要求每 100 个时钟周期触发一次 SysTick 中断，则将 RELOAD 设为 99。

表 89. SYST_RVR
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	RELOAD ：计数器达到 0 时加载至 SysTick 当前值寄存器的数值。	读写	0x000000

M0PLUS: SYST_CVR 寄存器

偏移量: 0xe018

描述

使用 SysTick 当前值寄存器以获取寄存器中的当前值。该寄存器的复位值未知。

表 90. SYST_CVR
寄存器

位	描述	类型	复位值
31:24	保留。	-	-

位	描述	类型	复位值
23:0	CURRENT: 读取时返回 SysTick 计数器的当前值。该寄存器为写入清零寄存器。向其写入任意值均会将寄存器清零为 0。清除此寄存器同时清除 SysTick 控制和状态寄存器中的 COUNTFLAG 位。	读写	0x000000

M0PLUS: SYST_CALIB 寄存器

偏移量: 0xe01c

描述

使用 SysTick 校准值寄存器，使软件能够通过除法和乘法缩放至任意所需速度。

表 91. SYST_CALIB 寄存器

位	描述	类型	复位值
31	NOREF: 若读取值为 1，表示未提供参考时钟——SysTick 控制和状态寄存器中的 CLKSOURCE 位将被强制为 1，且无法清零为 0。	只读	0x0
30	SKEW: 若读数为 1，则10ms 的校准值不准确（因时钟频率所致）。	只读	0x0
29:24	保留。	-	-
23:0	TENMS: 用于10ms (100Hz) 定时的可选重装载值，受系统时钟偏差误差影响。若数值为0，则表示校准值未知。	只读	0x000000

M0PLUS: NVIC_ISER 寄存器

偏移量: 0xe100

说明

使用中断置使能寄存器以启用中断，并确定当前已启用的中断。

若挂起的中断已被使能，NVIC 将根据其优先级激活该中断。若中断未被使能，断言其中断信号将使中断状态变为挂起，但无论优先级如何，NVIC 均不会激活该中断。

表 92. NVIC_ISER 寄存器

位	描述	类型	复位值
31:0	SETENA: 中断置使能位。 写入： 0 = 无效操作。 1 = 使能中断。 读取： 0 = 中断禁用。 1 = 中断使能。	读写	0x00000000

M0PLUS: NVIC_ICER 寄存器

偏移: 0xe180

描述

使用中断清除使能寄存器以禁用中断，并确定当前启用的中断。

表 93. NVIC_ICER 寄存器

位	描述	类型	复位值
31:0	CLRENA: 中断清除使能位。 写入： 0 = 无效操作。 1 = 禁用中断。 读取： 0 = 中断禁用。 1 = 中断使能。	读写	0x00000000

M0PLUS: NVIC_ISPR 寄存器

偏移: 0xe200

描述

NVIC_ISPR 强制中断进入待决状态，并显示哪些中断处于待决状态。

表 94. NVIC_ISPR 寄存器

位	描述	类型	复位值
31:0	SETPEND: 中断置待决位。 写入： 0 = 无效操作。 1 = 将中断状态更改为待决。 读取： 0 = 中断未处于待决状态。 1 = 中断处于待决状态。 注意：向 NVIC_ISPR 相关位写入 1 时： 一个已处于待决状态的中断无任何影响。 一个被禁用的中断会将该中断状态设置为待决。	读写	0x00000000

M0PLUS: NVIC_ICPR 寄存器

偏移: 0xe280

说明

使用中断清除待处理寄存器以清除待处理的中断，并确定当前哪些中断处于待处理状态。

表 95. NVIC_ICPR 寄存器

位	描述	类型	复位值
31:0	CLRPEND: 中断清除待处理位。 写入： 0 = 无效操作。 1 = 清除待处理状态及中断。 读取： 0 = 中断未处于待决状态。 1 = 中断处于待决状态。	读写	0x00000000

M0PLUS: NVIC_IPR0 寄存器

偏移: 0xe400

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

注意：向NVIC_ICPR位写入1不会影响对应中断的激活状态。

这些寄存器仅支持字访问。

表96. NVIC_IPR0
寄存器

位	描述	类型	复位值
31:30	IP_3: 中断3的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_2: 中断2的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_1: 中断1的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_0: 中断0的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR1寄存器

偏移: 0xe404

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 97. NVIC_IPR1
寄存器

位	描述	类型	复位值
31:30	IP_7: 中断7的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_6: 中断6的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_5: 中断5的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_4: 中断4的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR2 寄存器

偏移: 0xe408

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 98. NVIC_IPR2
寄存器

位	描述	类型	复位值
31:30	IP_11: 中断11的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_10: 中断10的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_9: 中断9的优先级	读写	0x0
13:8	保留。	-	-

位	描述	类型	复位值
7:6	IP_8: 中断8的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR3 寄存器

偏移: 0xe40c

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 99. NVIC_IPR3
寄存器

位	描述	类型	复位值
31:30	IP_15: 中断15的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_14: 中断14的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_13: 中断13优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_12: 中断12优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR4寄存器

偏移: 0xe410

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 100. NVIC_IPR4
寄存器

位	描述	类型	复位值
31:30	IP_19: 中断19优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_18: 中断18优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_17: 中断17优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_16: 中断16优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR5寄存器

偏移: 0xe414

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 101. NVIC_IPR5
寄存器

位	描述	类型	复位值
31:30	IP_23 : 中断23优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_22 : 中断22优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_21 : 中断21优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_20 : 中断20优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR6寄存器

偏移: 0xe418

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表102. NVIC_IPR6
寄存器

位	描述	类型	复位值
31:30	IP_27 : 中断27的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_26 : 中断26的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_25 : 中断25的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_24 : 中断24的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR7寄存器

偏移: 0xe41c

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表103. NVIC_IPR7
寄存器

位	描述	类型	复位值
31:30	IP_31 : 中断31的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_30 : 中断30的优先级	读写	0x0
21:16	保留。	-	-

位	描述	类型	复位值
15:14	IP_29 : 中断29的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_28 : 中断28的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: CPUID寄存器

偏移: 0xed00

描述

读取CPU ID基础寄存器以确定：处理器内核的ID编号、处理器内核的版本号及处理器内核的实现细节。

表104. CPUID
寄存器

位	描述	类型	复位值
31:24	IMPLEMENTER : 实现者代码: 0x41 = ARM	只读	0x41
23:20	VARIANT : rnpm修订状态中的主修订号n： 0x0 = 修订版0。	只读	0x0
19:16	ARCHITECTURE : 定义处理器架构的常量： 0xC = ARMv6-M架构。	只读	0xc
15:4	PARTNO : 系列内处理器编号: 0xC60 = Cortex-M0+	只读	0xc60
3:0	REVISION : rnpm修订状态中的次修订号m： 0x1 = 补丁1。	只读	0x1

M0PLUS: ICSR寄存器

偏移量: 0xed04

说明

使用中断控制状态寄存器设置挂起的不可屏蔽中断（NMI），设置或清除挂起的PendSV，设置或清除挂起的SysTick，检测挂起的异常，检查优先级最高的挂起异常向量号，检查当前活动异常向量号。

表105. ICSR
寄存器

位	描述	类型	复位值
31	NMIPENDSET : 设置此位将激活NMI。鉴于NMI为最高优先级异常，一经置位立即激活。 NMI 设定挂起位。 写： 0 = 无效操作。 1 = 将 NMI 异常状态更改为挂起。 读取： 0 = NMI 异常未挂起。 1 = NMI 异常已挂起。 由于NMI属于最高优先级异常，处理器通常在检测到对该位写入1时立即进入NMI 异常处理程序。进入该处理程序后，该位将被清除为0。 这意味着NMI异常处理程序读取该位时，只有在处理程序执行期间NMI信号被重新置位，才返回1。	读写	0x0

位	描述	类型	复位值
30:29	保留。	-	-
28	PENDSVSET : PendSV 挂起设置位。 写入： 0 = 无效操作。 1 = 将 PendSV 异常状态设为挂起。 读取： 0 = PendSV 异常未处于挂起状态。 1 = PendSV 异常处于挂起状态。 将该位写入 1 是设置 PendSV 异常状态为挂起的唯一方法。	读写	0x0
27	PENDSVCLR : PendSV 清除挂起位。 写入： 0 = 无效操作。 1 = 从 PendSV 异常中移除挂起状态。	读写	0x0
26	PENDSTSET : SysTick 异常置位挂起位。 写入： 0 = 无效操作。 1 = 将 SysTick 异常状态更改为挂起。 读取： 0 = SysTick 异常未处于挂起状态。 1 = SysTick 异常处于挂起状态。	读写	0x0
25	PENDSTCLR : SysTick 异常清除挂起位。 写入： 0 = 无效操作。 1 = 移除 SysTick 异常的挂起状态。 该位为只写 (WO)。寄存器读取时，其值为未知。	读写	0x0
24	保留。	-	-
23	ISRPREEMPT : 仅当核心停止时，系统方可访问此位。指示将在下一运行周期处理一个挂起中断。若调试停止控制与状态寄存器中的 C_MASKINTS 位清零，则中断将得到服务。	只读	0x0
22	ISR PENDING : 外部中断挂起标志。	只读	0x0
21	保留。	-	-
20:12	VECTPENDING : 指示最高优先级挂起异常的异常编号：0 = 无挂起异常。非零值 = 挂起状态包括内存映射的使能和屏蔽寄存器的效应。但不包括 PRIMASK 专用寄存器限定符的影响。	只读	0x000
11:9	保留。	-	-
8:0	VECTACTIVE : 活动异常编号字段。复位将清除 VECTACTIVE 字段。	只读	0x000

M0PLUS: VTOR寄存器

偏移: 0xed08

描述

VTOR寄存器保存向量表偏移地址。

表106. VTOR
寄存器

位	描述	类型	复位值
31:8	TBLOFF : 指示向量表偏移地址的[31:8]位。	读写	0x000000
7:0	保留。	-	-

M0PLUS: AIRCR寄存器

偏移: 0xed0c

描述

使用应用中断与复位控制寄存器以：确定数据字节序，清除调试暂停模式下的所有活动状态信息，发起系统复位请求。

表107. AIRCR
寄存器

位	描述	类型	复位值
31:16	VECTKEY : 寄存器密钥： 读取值为未知 写入时，须向VECTKEY写入0x05FA，否则写入操作将被忽略。	读写	0x0000
15	ENDIANESS : 实施的数据字节序： 0 = 小端序。	只读	0x0
14:3	保留。	-	-
2	SYSRESETREQ : 写1到该位将触发向外部系统发送SYSRESETREQ信号以请求复位。该操作旨在强制对除调试之外的所有主要组件进行系统复位。由于请求系统复位，DHCSR中的C_HALTI位被清除。调试器不会与设备失去连接。	读写	0x0
1	VECTCLRACTIVE : 清除所有固定和可配置异常的活动状态信息。 该位：自动清除，仅在核心停止时由DAP设置。设置时：清除处理器的所有活动异常状态，强制返回线程模式，并将IPSR强制置为0。调试器必须重新初始化堆栈。	读写	0x0
0	保留。	-	-

M0PLUS: SCR寄存器

偏移: 0xed10

描述

系统控制寄存器。用于电源管理功能：当处理器可进入低功耗状态时向系统发出信号，并控制处理器如何进入及退出低功耗状态。

表108. SCR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-

位	描述	类型	复位值
4	<p>SEVONPEND: 挂起时发送事件位： 0 = 仅启用的中断或事件可唤醒处理器，禁用的中断除外。 1 = 启用的事件及所有中断，包括禁用的中断，均可唤醒处理器。 当事件或中断处于待处理状态时，事件信号会唤醒处于WFE状态的处理器。 如果 处理器未处于等待事件状态，该事件将被登记并影响 下一次的WFE。 处理器也会因执行SEV指令或外部事件而唤醒。</p>	读写	0x0
3	保留。	-	-
2	<p>SLEEPDEEP: 控制处理器使用睡眠或深度睡眠作为低功耗模式： 0 = 睡眠。 1 = 深度睡眠。</p>	读写	0x0
1	<p>SLEEPONEXIT: 指示从处理程序模式返回线程模式时是否进入睡眠状态： 0 = 返回线程模式时不进入睡眠。 1 = 从中断服务程序返回线程模式时进入睡眠或深度睡眠。 将此位设为1，使基于中断的应用程序避免返回空闲的主程序。</p>	读写	0x0
0	保留。	-	-

M0PLUS: CCR 寄存器

偏移: 0xed14

描述

配置与控制寄存器永久启用栈对齐，并使非对齐访问导致硬错误（Hard Fault）。

表109. CCR
寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9	<p>STKALIGN: 始终读为1，表示异常进入时8字节栈对齐。异常进入时，处理器使用堆栈中PSR的第9位指示栈对齐。异常返回时，处理器利用该堆栈位恢复正确的栈对齐。</p>	只读	0x0
8:4	保留。	-	-
3	UNALIGN_TRP : 始终读为1，表示所有非对齐访问均触发硬错误。	只读	0x0
2:0	保留。	-	-

M0PLUS: SHPR2寄存器

偏移: 0xed1c

描述

系统处理程序是特殊的异常处理程序，可将其优先级设为任一等级。

使用系统处理器优先级寄存器2（System Handler Priority Register 2）设置SVCALL的优先级。

表110. SHPR2
寄存器

位	描述	类型	复位值
31:30	PRI_11 : 系统处理器11（SVCALL）的优先级	读写	0x0
29:0	保留。	-	-

M0PLUS: SHPR3寄存器

偏移: 0xed20

描述

系统处理器是特殊的异常处理器，可将其优先级设为任一等级。

使用系统处理器优先级寄存器3设置PendSV和SysTick的优先级。

表111. SHPR3
寄存器

位	描述	类型	复位值
31:30	PRI_15 : 系统处理器15（SysTick）的优先级	读写	0x0
29:24	保留。	-	-
23:22	PRI_14 : 系统处理器14（PendSV）的优先级	读写	0x0
21:0	保留。	-	-

M0PLUS: SHCSR寄存器

偏移: 0xed24

描述

使用系统处理器控制和状态寄存器以确定或清除SVCALL的挂起状态。

表112. SHCSR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	SVCALLPENDED : 若SVCALL挂起，则读值为1。写入1以设置SVCALL挂起，写入0以清除挂起状态。	读写	0x0
14:0	保留。	-	-

M0PLUS: MPU_TYPE寄存器

偏移: 0xed90

描述

读取MPU类型寄存器以确定处理器是否实现MPU及MPU支持的区域数量。

表113. MPU_TYPE
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	IREGION : 指令区域。读取值为零，因为ARMv6-M仅支持统一MPU。	只读	0x00
15:8	DREGION : MPU支持的区域数量。	只读	0x08
7:1	保留。	-	-

位	描述	类型	复位值
0	SEPARATE : 指示支持独立的指令和数据地址映射。读取值为0，因为ARMv6-M仅支持统一MPU。	只读	0x0

M0PLUS: MPU_CTRL 寄存器

偏移: 0xed94

描述

使用MPU控制寄存器启用或禁用MPU，并控制是否将默认内存映射作为特权访问的背景区域启用，以及是否在HardFault和NMI中启用MPU。

表114. MPU_CTRL
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	PRIVDEFENA : 控制是否将默认内存映射作为特权软件访问的背景区域启用。当ENABLE位清零时，该位被忽略。 0 = 若启用MPU，则禁用默认内存映射。对未被任何启用区域覆盖的位置的任何内存访问均会导致故障 。 1 = 若启用MPU，则允许使用默认内存映射作为特权软件访问的背景区域。 启用时，背景区域视为区域编号 -1。任何已定义且启用的区域均优先于此默认映射。	读写	0x0
1	HFNMIENA : 控制 MPU 在 HardFault 和 NMI 中断期间的使用。若 ENABLE 位为清除状态时设置此位，将导致不可预测的行为。 当 MPU 启用时： 0 = 无论 ENABLE 位的值，MPU 在 HardFault 和 NMI 处理程序期间均被禁用。 1 = MPU 在 HardFault 和 NMI 处理程序期间启用。	读写	0x0
0	ENABLE : 启用 MPU。若 MPU 被禁用，特权和非特权访问将使用默认内存映射。 0 = MPU 禁用。 1 = MPU 启用。	读写	0x0

M0PLUS: MPU_RNR 寄存器

偏移: 0xed98

描述

使用 MPU 区域编号寄存器选择当前由 MPU_RBAR 和 MPU_RASR 访问的区域。

表 115. MPU_RNR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	REGION : 指示 MPU_RBAR 和 MPU_RASR 寄存器所引用的 MPU 区域 。 MPU 支持 8 个内存区域，因此此字段允许的值为 0-7。	读写	0x0

M0PLUS: MPU_RBAR 寄存器

偏移: 0xed9c

描述

读取 MPU 区域基地址寄存器以确定由 MPU_RNR 标识区域的基地址。
写入以更新该区域或指定区域的基地址，同时 MPU_RNR 也将更新为该区域编号。

表 116. MPU_RBAR
寄存器

位	描述	类型	复位值
31:8	ADDR: 区域的基地址。	读写	0x000000
7:5	保留。	-	-
4	VALID: 写入时，指示写入是否必须更新 REGION 字段标识区域的基地址，并更新 MPU_RNR 以指示该新区域。 写入： 0 = MPU_RNR 不变，处理器： 更新 MPU_RNR 指定区域的基地址。 忽略 REGION 字段的值。 1 = 处理器： 将 MPU_RNR 更新为 REGION 字段的值。 更新 REGION 字段指定区域的基地址。 始终读取为零。	读写	0x0
3:0	REGION: 写入时指定区域编号，用以更新基地址，条件是 VALID 被写为1。读取时，返回 MPU_RNR 的第 [3:0] 位。	读写	0x0

M0PLUS: MPU_RASR 寄存器

偏移: 0xedaa0

说明

使用 MPU 区域属性及大小寄存器定义由 MPU_RNR 指定区域的大小、访问行为及内存类型，并启用该区域。

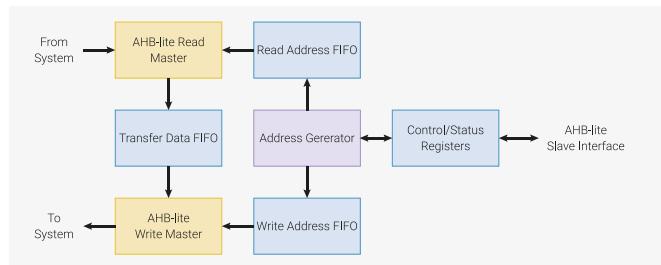
表 117. MPU_RASR
寄存器

位	描述	类型	复位值
31:16	ATTRS: MPU 区域属性字段。用于定义区域属性控制。 28 = XN：指令访问禁用位： 0 = 允许指令取指。 1 = 禁止指令取指。 26:24 = AP：访问权限字段 18 = S：共享位 17 = C：可缓存位 16 = B：缓冲位	读写	0x0000
15:8	SRD: 子区域禁用。对于大小为256字节及以上的区域，此字段的每个位控制八个等分子区域中的一个是否启用。	读写	0x00
7:6	保留。	-	-
5:1	SIZE: 表示区域大小。区域大小（字节）= $2^{(SIZE+1)}$ 。允许的最小值为7（b00111）= 256字节。	读写	0x00
0	ENABLE: 启用该区域。	读写	0x0

2.5. DMA

RP2040直接内存访问（DMA）控制器具备独立的读写主控接口，连接至总线结构，并代表处理器执行大容量数据传输。这使处理器可以自由处理其他任务，或进入低功耗睡眠状态。DMA的数据吞吐量显著高于RP2040的单个处理器。

图12。 DMA
架构概述。
读主控每个时钟周期均可从某地址读取数据。同样，写主控可以向另一个地址写入数据。
地址生成器产生配对的读写地址，主控通过地址FIFO使用这些地址。最多可同时进行12个传输序列，由软件通过控制与状态寄存器监督。



DMA每个时钟周期可执行一次最多32位的数据读访问和一次写访问。共有12个独立通道，每个通道负责监督一系列总线传输，通常适用于以下场景之一：

- 内存至外设：当外设需要更多数据进行传输时，会向DMA发出信号。DMA从RAM或闪存中的数组读取数据，并写入外设的数据FIFO。
- 外设至内存：当外设接收到数据时，会向DMA发出信号。DMA从外设的数据FIFO读取该数据，并写入RAM中的数组。
- 内存至内存：DMA在RAM的两个缓冲区之间尽可能快速地传输数据。

每个通道都配有独立的控制和状态寄存器（CSRs），软件可以通过它们对通道进行编程和进度监控。当多个通道同时激活时，DMA会在所有正在请求数据传输的通道间均匀分配带宽，采用轮询方式。

传输的数据大小可以是32位、16位或8位。每个通道仅需配置一次：源传输大小与目标传输大小保持一致。DMA在窄幅写入时执行标准字节通道复制，因此字节数据在数据总线的所有4个字节均可用，半字数据则在两个半字处均可用。

通道可通过多种方式组合，以实现更为复杂的行为和更高的自主性。例如，一个通道可以配置另一个通道，借助内存中一系列控制块加载配置信息；第二个通道在需要重新配置时，可通过 [CHAIN_TO](#) 选项回调第一个通道。

提高DMA的自主性意味着处理器监督需求大幅减少：总体而言，这使系统能够同时执行更多任务，或有效降低功耗。

2.5.1. 通道配置

每个通道包含四个控制/状态寄存器：

- **READ_ADDR** 指向下一次读取的地址
- **WRITE_ADDR** 指向下一次写入的地址
- **TRANS_COUNT** 显示当前传输序列中剩余的传输次数，并用于设定下一传输序列的传输次数（详见第2.5.1.2节）。
- **CTRL** 用于配置通道行为的所有其他方面，以启用或禁用该通道，并检查其完成状态。

这些为实时寄存器：它们会随通道的运行不断更新。

2.5.1.1. 读写地址

`READ_ADDR`和`WRITE_ADDR`分别包含通道下一步将读取和写入的地址。这些寄存器会在每次读写访问后自动更新。根据`CTRL`中配置的传输大小，它们每次递增1、2或4字节。

软件通常应在每次新的传输序列开始时，向这些寄存器写入新的起始地址。

若`READ_ADDR`和`WRITE_ADDR`未重新编程，DMA将使用当前值作为下一次传输的起始地址。例如：

- 如果地址不递增（例如地址指向外设FIFO），且下一传输序列也是来自或写入该相同地址，则无需再次写入该寄存器。
- 在内存中传输连续缓冲区序列（例如分散和聚集）时，地址寄存器将在一次传输完成时自动递增至下一个缓冲区的起始地址。

通过不为每个传输序列编程全部四个CSR，软件可使用更短的中断处理程序，并在使用通道链时采用更紧凑的控制块格式（详见第2.5.2.1节的寄存器别名及第2.5.2.2节的链式传输）。

⚠ 注意

`READ_ADDR`和`WRITE_ADDR`必须始终与当前传输大小对齐，该传输大小由`CTRL.DATA_SIZE`指定。软件需确保初始值的正确对齐。

2.5.1.2. 传输计数

读取`TRANS_COUNT`可获知当前传输序列中剩余的传输次数。该数值会随着通道传输的进行持续更新。写入`TRANS_COUNT`将设置下一次传输序列的长度。单个序列中最多可执行 2^{13} - 1次传输。

每当通道启动新的传输序列时，最近写入`TRANS_COUNT`的值会被复制到实时传输计数器，随后该计数器将随着新传输序列的进展开始递减。出于调试目的，可从`DBG_TCR`（`TRANS_COUNT`重载值）寄存器读取最后写入的值。

如果通道在未对`TRANS_COUNT`进行写入的情况下被多次触发，则每次都会执行相同次数的传输。例如，当串联时，一个通道可能会把固定大小的控制块加载到另一个通道的CSR中。软件仅需编程一次`TRANS_COUNT`，然后每次自动重载。

或者，可以在每次启动传输序列之前向`TRANS_COUNT`写入新的值。如果`TRANS_COUNT`是通道触发器（见第2.5.2.1节），通道将立即启动，并使用刚写入的值，而非当前重载寄存器中的值。

ℹ 注意

`TRANS_COUNT`表示要执行的传输次数。传输的总字节数为`TRANS_COUNT`乘以每次传输的字节大小，该大小由`CTRL.DATA_SIZE`指定。

2.5.1.3. 控制/状态

与其他三个寄存器相比，`CTRL`寄存器包含更多且更小的字段，详细信息请参见`CTRL`寄存器列表。除其他用途外，`CTRL`用于：

- 通过`CTRL.DATA_SIZE`配置该通道数据传输的大小。读写操作的大小相同。
- 通过`CTRL.INCR_WRITE`、`CTRL.INCR_READ`、`CTRL.RING_SEL`及`CTRL.RING_SIZE`配置是否以及如何在每次读写后递增`READ_ADDR`和`WRITE_ADDR`。支持环形传输，其中一个地址指针在某个2的幂次方边界处回绕。

- 通过`CTRL.CHAIN_TO`选择另一通道（或无通道）以在本通道完成后触发。
- 通过`CTRL.TREQ_SEL`选择一个外设数据请求（DREQ）信号以控制该通道的传输节奏。
- 通过`CTRL.BUSY`查看通道是否空闲。
- 通过`CTRL.AHB_ERROR`、`CTRL.READ_ERROR`或`CTRL.WRITE_ERROR`查看通道是否发生总线错误，例如访问了错误地址。

2.5.2. 启动通道

启动通道有三种方式：

- 向通道触发寄存器写入数据
- 从刚完成的另一个通道触发链式启动，且其`CHAIN_TO`字段已配置
- 使用`MULTI_CHAN_TRIGGER`寄存器，可同时启动多个通道

每种方式适用于不同的使用场景。例如，触发寄存器在中断服务例程中配置并启动通道时表现出简单且高效的特点，而`CHAIN_TO`允许一个通道回调另一个通道，后者随后可重新配置第一个通道。

① 注意

触发已运行中的通道无任何效果。

2.5.2.1. 别名与触发器

表118。控制寄存器别名每个通道具有四个控制/状态寄存器。每个寄存器可以通过多个不同地址访问。在每组自然对齐的四个寄存器中，所有四个寄存器都会出现，但顺序不同。

偏移量	+0x0	+0x4	+0x8	+0xC (触发)
0x00 (别名0)	READ_ADDR	WRITE_ADDR	TRANS_COUNT	CTRL_TRIG
0x10 (别名1)	CTRL	READ_ADDR	WRITE_ADDR	TRANS_COUNT_TRIG
0x20 (别名2)	CTRL	TRANS_COUNT	READ_ADDR	WRITE_ADDR_TRIG
0x30 (别名3)	CTRL	WRITE_ADDR	TRANS_COUNT	READ_ADD_TRIG

这四个控制/状态寄存器在内存中被多重别名映射。每个别名一共计四个一一映射相同的四个物理寄存器，但顺序不同。每个别名中最后一个寄存器（偏移+0xC，已高亮）为触发寄存器。

向触发寄存器写入数据将启动该通道。

通常仅使用别名0，别名1至3可忽略。通过写入`READ_ADDR`、`WRITE_ADDR`、`TRANS_COUNT`，最后写入`CTRL`来配置并启动通道。由于`CTRL`是别名0中的触发寄存器，因此此操作启动通道。

其他别名允许在使用一个通道配置另一个时，实现更紧凑的控制块列表，以及在中断处理程序中更高效的重新配置与启动：

- 每个CSR均为某别名中的触发寄存器：
 - 在将固定大小缓冲区聚集到外设时，可仅通过写入`READ_ADDR_TRIG`配置并启动DMA通道。
 - 在外设分散到固定大小缓冲区时，可仅通过写入`WRITE_ADDR_TRIG`配置并启动通道。
- 寄存器的有效组合呈现为自然对齐的元组，其中包含一个触发寄存器。结合通道链和地址环绕，可实现压缩控制块格式，如：
 - (`WRITE_ADDR`, `TRANS_COUNT_TRIG`) 用于外设散射操作

- (TRANS_COUNT, READ_ADDR_TRIG) 用于外设聚合操作，或对缓冲区列表计算 CRC
- (READ_ADDR, WRITE_ADDR_TRIG) 用于操作内存中的固定大小缓冲区

触发寄存器在以下情况下不会启动通道：

- 通道通过 CTRL.EN 被禁用。（若触发源为 CTRL，使用刚写入的 EN 值，而非 CTRL 寄存器中当前的值。）
- 通道已在运行中。
- 触发寄存器写入值为 0。（此功能用于结束控制块链。详见第 2.5.2.3 节“空触发”）

2.5.2.2. 链接

当通道完成时，可以指定另一个通道立即触发。此功能可用作第二通道的回调，用于重新配置并重启第一个通道。

该功能通过通道 CTRL 寄存器中的 CHAIN_TO 字段进行配置。该 4 位数值选择一个通道，当本通道结束时，将启动所选通道。通道不可链至自身。将 CHAIN_TO 设置为通道自身的索引表示不会发生链式连接。

链式触发的行为与来自其他源（例如触发寄存器）的触发相同。例如，它们会导致 TRANS_COUNT 重新加载，且若目标通道已运行则会被忽略。

CHAIN_TO 的一个应用是，通道请求由另一通道从内存中的控制块序列进行重新配置。通道 A 配置为从内存执行环绕传输至通道 B 的控制寄存器（包括触发寄存器），通道 B 配置为在完成每个传输序列后链回通道 A。此情形在 DMA 控制块示例（第 2.5.6.2 节）中有更明确的说明。

使用寄存器别名（第 2.5.2.1 节）可实现 DMA 控制块的紧凑格式，在某些情况下仅需一个字。

链式使用的另一种形式是“乒乓”配置，其中两个通道相互触发。处理器可以响应通道完成的中断，并在通道完成后重新配置该通道；然而，已配置完毕的链式通道会立即开始。换言之，通道配置与通道操作实现流水线处理。在需要大量短传输序列的情况下，性能可显著提升。

第 2.5.6 节详细说明了链式触发在实际应用中的可能性。

2.5.2.3. 空触发与链式中断

如第 2.5.2.1 节所述，向触发寄存器写入全零值不会启动通道。此称为空触发，具有以下两个目的：

- 通过添加全零控制块，实现控制块数组末尾的停止
- 减少使用控制块时生成的中断数量

默认情况下，每当通道完成传输序列时，都会产生中断，除非该通道的 IRQ 在 INTE0 或 INTE1 中被屏蔽。中断的频率可能过高，尤其在一系列控制块执行期间，处理器通常无需关注；但在链的末端，处理器关注是必要的。

通道 CTRL 寄存器包含一个名为 IRQ_QUIET 的字段，其默认值为 0。当该字段设置为 1 时，通道仅在接收到空触发信号时触发中断，其他情况下不触发中断。中断由接收到触发信号的通道产生。

2.5.3. 数据请求 (DREQ)

外设以自身速率产生或消耗数据。若DMA仅按最大速度传输数据，将导致数据丢失或损坏。DREQ是外设与DMA之间的通信通道，使DMA能根据外设需求调整传输节奏。

`CTRL.TREQ_SEL`（传输请求）字段用于选择外部DREQ，也可选择内部节奏定时器之一，或选择不使用TREQ（传输以最大速度进行），例如内存到内存传输。

2.5.3.1. 系统 DREQ 表

DREQ 编号被全局分配至外设 DREQ 通道。

表 119. DREQs

DREQ	DREQ 通道	DREQ	DREQ 通道	DREQ	DREQ 通道	DREQ	DREQ 通道
0	DREQ_PI00_TX0	10	DREQ_PI01_TX2	20	DREQ_UART0_TX	30	DREQ_PWM_WRAP6
1	DREQ_PI00_RX1	11	DREQ_PI01_RX3	21	DREQ_UART0_RX	31	DREQ_PWM_WRAP7
2	DREQ_PI00_RX2	12	DREQ_PI01_RX0	22	DREQ_UART1_TX	32	DREQ_I2C0_RX
3	DREQ_PI00_RX3	13	DREQ_PI01_RX1	23	DREQ_UART1_RX	33	DREQ_I2C0_RX
4	DREQ_PI00_RX0	14	DREQ_PI01_RX2	24	DREQ_PWM_WRAP0	34	DREQ_I2C1_RX
5	DREQ_PI00_RX1	15	DREQ_PI01_RX3	25	DREQ_PWM_WRAP1	35	DREQ_I2C1_RX
6	DREQ_PI00_RX2	16	DREQ_SPI0_TX	26	DREQ_PWM_WRAP2	36	DREQ_ADC
7	DREQ_PI00_RX3	17	DREQ_SPI0_RX	27	DREQ_PWM_WRAP3	37	DREQ_XIP_STREAM
8	DREQ_PI01_TX0	18	DREQ_SPI1_TX	28	DREQ_PWM_WRAP4	38	DREQ_XIP_SSITX
9	DREQ_PI01_RX1	19	DREQ_SPI1_RX	29	DREQ_PWM_WRAP5	39	DREQ_XIP_SSIRX

2.5.3.2. 基于信用的 DREQ 方案

RP2040 DMA 设计适用于以下系统：

- 大型外设数据 FIFO 在面积和功耗方面的成本过高
- 个别外设的带宽需求可能较高，例如短时总线注入率超过 50%
- 总线延迟较低，但多个主控可能竞争总线访问权限

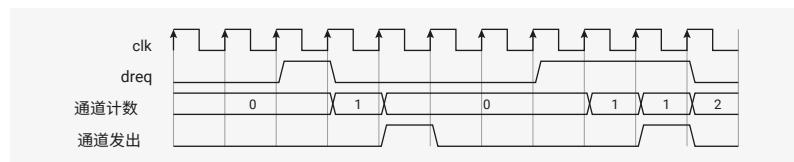
此外，DMA 的传输 FIFO 及双主控结构允许同一外设同时存在多个访问请求，从而提升整体吞吐量。因此，DREQ 机制的选择至关重要：

- 传统的“开启水龙头”方式若多个写操作在 TDF 中积压，可能导致溢出。一些系统通过为外设 FIFO 过度配置容量并将 DREQ 阈值设置在满容量以下来解决此问题，但这将浪费宝贵的面积和功耗资源。
- ARM风格的单次和突发握手协议在当前请求处理期间不允许注册额外请求。当FIFO较浅时，此机制会限制性能表现。

RP2040 DMA采用基于信用的DREQ机制。对于每个外设，DMA尝试维持尽可能多的传输同时进行，数量取决于外设的处理能力。此举实现了通过具有8级深度外设FIFO的全总线吞吐量（每时钟周期1字），在不存在布线延迟或资源竞争的情况下，避免FIFO发生溢出或欠载。

DMA为每个通道维护一个计数器。每个 `dreq`信号的1时钟脉冲会使该计数器递增（饱和计数）。计数器非零时，通道向DMA内部仲裁器请求传输；当传输下发到地址FIFO时计数器递减。此阶段传输处于进行中，但尚未完成。

图13. DREQ
计数



其作用是在外设FIFO中可用的空间或数据量基础上，对在途传输数量设定上限。在稳态下，这能实现最大吞吐量，且不会发生溢出或欠载。

一项注意事项是，用户不得访问当前正由DMA服务的FIFO。这将导致通道与外设不同步，可能引发数据损坏或丢失。

另一项注意事项是，不应将多个通道连接至同一DREQ。

2.5.4. 中断

每个通道均可产生中断；这些中断可通过 `INTE0` 或 `INTE1` 寄存器按通道掩码。

通道在以下两种情况下会触发中断请求：

- 每个传输序列完成时，且 `CTRL.IRQ QUIET` 被禁用时
- 接收到空触发信号时，且 `CTRL.IRQ QUIET` 被启用时

被掩码的中断状态可在INTS寄存器中查看；该寄存器为每个通道对应一个位。通过向INTS写入位掩码以清除中断。确认中断的一种惯用做法是先读取INTS，再将相同的值写回，如此仅会清除已启用的中断。

RP2040 DMA 提供两个系统 IRQ，具有独立的屏蔽和状态寄存器（例如 `INTE0` 和 `INTE1`）。任意组合的通道中断请求均可路由至任一系统 IRQ。例如：

- 若某些通道具有特别严格的时序要求，则可在系统中断控制器中赋予其更高优先级。
- 在多处理器系统中，不同通道的中断请求可独立路由至不同核心。

出于调试目的，INTF寄存器可强制触发任一IRQ的断言。

2.5.5. 附加功能

2.5.5.1. 节奏计时器

该计时器允许大致每隔 n `clk_sys` 时钟周期传输一次数据，无需通过外部外设的 DREQ 触发传输。采用分数 (X/Y) 分频器设计，每个 `clk_sys` 时钟周期最多可产生一次请求。

RP2040 中提供 4 个定时器。每个 DMA 可在 `CTRL.TREQ_SEL` 字段中选择其中任意一个。

2.5.5.2. CRC 计算

DMA 可监视指定通道经数据 FIFO 传递的数据，并基于此数据计算校验和。此过程完全被动：硬件不会修改数据，仅进行监测。

该功能由 `SNIFF_CTRL` 和 `SNIFF_DATA` 寄存器控制，且可通过 `CTRL.SNIFF_EN` 字段在每次 DMA 传输时启用或禁用。

由于此硬件无法对 FIFO 施加回压，故必须满足 DMA 最大 32 位/时钟的传输速率要求。

支持的校验和类型如下：

- CRC-32，最高有效位优先及最低有效位优先
- CRC-16-CCITT，最高有效位优先及最低有效位优先
- 简单求和（累计至32位累加器）
- 偶数奇偶校验

结果寄存器支持读写，允许设置初始种子值。

结果支持位/字节操作，可能有助于特定用例：

- 位反转
- 位逆转
- 字节交换

这些操作不会影响CRC计算，仅改变结果寄存器中数据的呈现方式。

2.5.5.3. 通道中止

通道可能进入不可恢复状态：例如，如果指令传输的数据量超出外设请求的范围，传输将永远无法完成。清除 `CTRL.EN` 位仅会暂停通道，不能解决该问题。此情况在正常操作中不应发生，但必须具备无需硬重置整个DMA模块即可恢复的机制。

`CHAN_ABORT` 寄存器可强制通道提前终止。寄存器中每个位对应一条通道，写入1则终止相应通道。此操作会清除传输计数器，并将通道置于非活动状态。

⚠ 注意

由于RP2040-E13问题，终止正在传输数据（即未因DREQ非活动而阻塞）的DMA通道，可能会触发完成中断（IRQ）。执行中止操作前，应清除通道中断使能位；中止操作后，应检查并清除中断。

触发中止时，通道可能存在读写主设备间正在进行的总线传输，且这些传输不可撤销。`CTRL.BUSY` 标志将保持高电平，直到所有传输完成且通道进入安全状态，通常仅需数个时钟周期。通道必须等 `CTRL.BUSY` 标志复位后，方可重新启动。在旧序列传输尚未完成时启动新序列的传输，可能导致不可预测行为。

2.5.5.4. 调试

每个DMA通道配备调试寄存器，用于显示dreq计数器 `BG_CTDREQ` 及下一个传输计数 `BG_TCR`。如有需要，这些寄存器也可用于复位DMA通道。

2.5.6. 示例用例

2.5.6.1. 使用中断重新配置通道

通道完成一组传输后，即可用于执行后续更多传输。软件检测到通道不再忙碌，随后重新配置并重启该通道。一种方法是轮询 `CTRL_BUSY` 位，直到通道完成，但这会失去DMA的一个关键优势，即其无需与处理器同步操作。通过设置 `INTE0` 或 `INTE1` 中的相应位，我们可以指示DMA在某个通道完成时激活其两条中断请求线之一。我们无需反复查询通道是否完成，而是直接收到通知。

ⓘ 注意

使用两条系统中断线，允许将不同通道的完成中断路由到不同核心，或在同一核心上根据通道的时间紧迫性实现中断抢占。

当中断被触发时，处理器可以配置为暂停当前操作，调用用户指定的处理程序函数。处理程序可以重新配置并重启该通道。处理程序结束后，处理器恢复执行被中断的前台代码。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c 第35至52行

```

35 void dma_handler() {
36     static int pwm_level = 0;
37     static uint32_t wavetable[N_PWM_LEVELS];
38     static bool first_run = true;
39     // 条目编号 `i` 含有 `i` 个1位及 `(32 - i)` 个0位。
40     if (first_run) {
41         first_run = false;
42         for (int i = 0; i < N_PWM_LEVELS; ++i)
43             wavetable[i] = ~(~0u << i);
44     }
45
46     // 清除中断请求。
47     dma_hw->ints0 = 1u << dma_chan;
48     // 为该通道分配新的波形表条目以供读取，并重新触发其操作
49     dma_channel_set_read_addr(dma_chan, &wavetable[pwm_level], true);
50
51     pwm_level = (pwm_level + 1) % N_PWM_LEVELS;
52 }
```

在多数情况下，大部分配置可在通道首次启动时完成，仅需在DMA处理程序中重新编程地址及传输长度。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c 第54至94行

```

54 int main() {
55 #ifndef PICO_DEFAULT_LED_PIN
56 // warning dma/channel_irq 示例要求开发板配备常规LED
57 #else
58     // 设置PIO状态机以序列化位流
59     uint offset = pio_add_program(pio0, &pio_serialiser_program);
60     pio_serialiser_program_init(pio0, 0, offset, PICO_DEFAULT_LED_PIN, PIO_SERIAL_CLKDIV);
61
62     // 配置通道以反复向PIO0写入相同的32位数据
63     // SM0 的 TX FIFO，由该外设的数据请求信号控制节奏。
64     dma_chan = dma_claim_unused_channel(true);
65     dma_channel_config c = dma_channel_get_default_config(dma_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, false);
68     channel_config_set_dreq(&c, DREQ_PIO0_TX0);
69
70     dma_channel_configure(
71         dma_chan,
72         &c,
73         &pio0_hw->txf[0], // 写入地址（仅需设置一次）
74         NULL, // 暂不提供读取地址
75         PWM_REPEAT_COUNT, // 多次写入相同值，随后暂停并触发中断
76         false // 暂不启动
77     );
78 }
```

```

79 // 指示 DMA 在通道完成数据块时触发 IRQ 0 线
80 dma_channel_set_irq0_enabled(dma_chan, true);
81
82 // 配置处理器在 DMA IRQ 0 触发时执行 dma_handler()
83 irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
84 irq_set_enabled(DMA_IRQ_0, true);
85
86 // 手动调用处理程序一次，以触发首次传输
87 dma_handler();
88
89 // 此后所有操作均由中断驱动。处理器有
90 // 时间思考其提前退休的计划——或许开一家面包店？
91 while (true)
92     tight_loop_contents();
93 #endif
94 }

```

此技术的一个缺点在于，必须等到通道完成最后一次传输后，才能开始重新配置通道。若处理器承受频繁的中断活动，该延迟可能相当长，导致传输间断较大，这对于需要维持高数据吞吐量的场景而言存在问题。

该问题通过使用两个通道及其 `CHAIN_TO` 字段交叉配置得以解决，即通道A完成后触发通道B，反之亦然。在任何时刻，其中一个通道正在传输数据，另一个通道要么已配置为在当前传输结束后立即启动下一次传输，要么正处于重新配置过程。当通道A完成传输时，会立即启动通道B上排队的传输。与此同时，中断被触发，处理程序会重新配置通道A，以便在通道B完成时做好准备。

2.5.6.2. DMA 控制块

通常，需要将多个较小缓冲区汇总并发送至同一外设。为满足此类需求，RP2040 DMA 可实现无需处理器干预的长且复杂的传输序列。一个通道反复重新配置第二个通道，第二个通道每完成一块传输后则重新启动第一个通道。

由于第一个DMA通道直接从内存向第二个通道的控制寄存器传输数据，内存中控制块的格式必须与这些寄存器保持一致。每次写入的最后一个寄存器，将是触发寄存器之一（第2.5.2.1节），它将启动第二通道执行其已编程的传输块。寄存器别名（第2.5.2.1节）为块布局提供了灵活性，更重要的是允许从块中省略某些寄存器，从而减少内存占用并提高加载速度。

此示例展示了如何通过重新编程 `TRANS_COUNT` 和 `READ_ADDR_TRIGGER` 来收集多个缓冲区并传输至UART：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/control_blocks/control_blocks.c

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 // 使用两个DMA通道构成一个编程的数据传输序列，传输至
8 // UART (数据收集操作)。一个通道负责传输
9 // 实际数据，另一个通道则不断重新编程该通道。
10
11 #include <stdio.h>
12 #include "pico/stlolib.h"
13 #include "hardware/dma.h"
14 #include "hardware/structs/uart.h"

```

```

15
16 // 这些缓冲区将依次通过DMA传输至UART。
17
18 const char word0[] = "Transferring ";
19 const char word1[] = "one ";
20 const char word2[] = "word ";
21 const char word3[] = "at ";
22 const char word4[] = "a ";
23 const char word5[] = "time.\n";
24
25 // 注意字段的顺序：长度字段必须位于读取地址之前，
26 // 因为控制通道将向数据通道别名3中最后两个寄存器写入：
27 // two registers in alias 3 on the data channel:
28 //          +0x0      +0x4      +0x8      +0xC (Trigger)
29 // 别名 0: READ_ADDR  WRITE_ADDR TRANS_COUNT CTRL
30 // 别名 1: CTRL      READ_ADDR   WRITE_ADDR  TRANS_COUNT
31 // 别名 2: CTRL      TRANS_COUNT READ_ADDR   WRITE_ADDR
32 // 别名 3: CTRL      WRITE_ADDR   TRANS_COUNT READ_ADDR
33 //
34 // 这将设置数据通道的传输计数和读取地址，
35 // 并触发该通道。数据通道完成后，将重新启动
36 // 控制通道（通过 CHAIN_TO），以加载接下来的两个字到其控制
37 // 寄存器中。
38
39 const struct {uint32_t len; const char *data;} control_blocks[] = {
40     {count_of(word0) - 1, word0}, // 跳过空终止符
41     {count_of(word1) - 1, word1},
42     {count_of(word2) - 1, word2},
43     {count_of(word3) - 1, word3},
44     {count_of(word4) - 1, word4},
45     {count_of(word5) - 1, word5},
46     {0, NULL}                  // 用于结束链的空触发器。
47 };
48
49 int main() {
50 #ifndef uart_default
51 #warning dma/control_blocks 示例需要UART
52 #else
53     stdio_init_all();
54     puts("DMA 控制块示例: ");
55
56     // ctrl_chan 将控制块加载到 data_chan，后者负责执行它们。
57     int ctrl_chan = dma_claim_unused_channel(true);
58     int data_chan = dma_claim_unused_channel(true);
59
60     // 控制通道将两个字传输到数据通道的控制
61     // 寄存器，然后停止。写入地址在两个字的
62     // (八字节) 边界处循环，因此控制通道会写入相同的两个字
63     // 下次触发时的寄存器数量。
64
65     dma_channel_config c = dma_channel_get_default_config(ctrl_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, true);
68     channel_config_set_write_increment(&c, true);
69     channel_config_set_ring(&c, true, 3); // 写指针的 1 << 3 字节边界
70
71     dma_channel_configure(
72         ctrl_chan,
73         &c,
74         &dma_hw->ch[data_chan].a13_transfer_count, // 初始写地址
75         &control_blocks[0],                          // 初始读地址
76         2,                                         // 每个控制块后暂停
77         false                                      // 暂不启动
78     );

```

```

79
80 // 数据通道配置为写入 UART FIFO (由 UART 的 TX 数据请求信号控制)
81 // 完成后再链接至控制通道
82 // once it completes. 控制通道设置新的读取地址及
83 // 数据长度，并重新触发数据通道。
84
85 c = dma_channel_get_default_config(data_chan);
86 channel_config_set_transfer_data_size(&c, DMA_SIZE_8);
87 channel_config_set_dreq(&c, uart_get_dreq(uart_default, true));
88 // 当 data_chan 完成时，触发 ctrl_chan
89 channel_config_set_chain_to(&c, ctrl_chan);
90 // 向触发寄存器写入 0 (链结束) 时，提升 IRQ 标志：
91 channel_config_set_irq_quiet(&c, true);
92
93 dma_channel_configure(
94     data_chan,
95     &c,
96     &uart_get_hw(uart_default)->dr,
97     NULL,           // 初始读取地址和传输计数无关紧要;
98     0,              // 控制通道将每次重新编程它们。
99     false           // 暂不启动。
100 );
101
102 // 一切准备就绪，通知控制通道加载首个
103 // 控制块。此后操作全部自动完成。
104 dma_start_channel_mask(1u << ctrl_chan);
105
106 // 当数据通道收到空触发时，将断言其 IRQ 标志，
107 // 表示控制块列表结束。我们将等待 108 // IRQ 标志，而不设置中断处理程序。
108
109 while (!(dma_hw->intr & 1u << data_chan))
110     tight_loop_contents();
111 dma_hw->ints0 = 1u << data_chan;
112
113 puts("DMA 完成。");
114 #endif
115 }

```

2.5.7. 寄存器列表

DMA 寄存器起始于基地址 **0x50000000**(在 SDK 中定义为 DMA_BASE)。

表 120。DMA 寄存器列表

偏移量	名称	说明
0x000	CH0_READ_ADDR	DMA 通道 0 读地址指针
0x004	CH0_WRITE_ADDR	DMA 通道 0 写地址指针
0x008	CH0_TRANS_COUNT	DMA 通道 0 传输计数
0x00c	CH0_CTRL_TRIG	DMA 通道 0 控制与状态寄存器
0x010	CH0_AL1_CTRL	通道 0 CTRL 寄存器的别名
0x014	CH0_AL1_READ_ADDR	通道 0 READ_ADDR 寄存器的别名
0x018	CH0_AL1_WRITE_ADDR	通道 0 WRITE_ADDR 寄存器的别名
0x01c	CH0_AL1_TRANS_COUNT_TRIG	通道 0 TRANS_COUNT 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x020	CH0_AL2_CTRL	通道 0 CTRL 寄存器的别名

偏移量	名称	说明
0x024	CH0_AL2_TRANS_COUNT	通道 0 TRANS_COUNT 寄存器的别名
0x028	CH0_AL2_READ_ADDR	通道 0 READ_ADDR 寄存器的别名
0x02c	CH0_AL2_WRITE_ADDR_TRIG	通道 0 WRITE_ADDR 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x030	CH0_AL3_CTRL	通道 0 CTRL 寄存器的别名
0x034	CH0_AL3_WRITE_ADDR	通道 0 WRITE_ADDR 寄存器的别名
0x038	CH0_AL3_TRANS_COUNT	通道 0 TRANS_COUNT 寄存器的别名
0x03c	CH0_AL3_READ_ADDR_TRIG	通道 0 READ_ADDR 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x040	CH1_READ_ADDR	DMA 通道 1 读地址指针
0x044	CH1_WRITE_ADDR	DMA 通道 1 写地址指针
0x048	CH1_TRANS_COUNT	DMA 通道 1 传输计数
0x04c	CH1_CTRL_TRIG	DMA 通道 1 控制与状态寄存器
0x050	CH1_AL1_CTRL	通道 1 CTRL 寄存器别名
0x054	CH1_AL1_READ_ADDR	通道 1 READ_ADDR 寄存器别名
0x058	CH1_AL1_WRITE_ADDR	通道 1 WRITE_ADDR 寄存器别名
0x05c	CH1_AL1_TRANS_COUNT_TRIG	通道 1 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x060	CH1_AL2_CTRL	通道 1 CTRL 寄存器别名
0x064	CH1_AL2_TRANS_COUNT	通道 1 TRANS_COUNT 寄存器别名
0x068	CH1_AL2_READ_ADDR	通道 1 READ_ADDR 寄存器别名
0x06c	CH1_AL2_WRITE_ADDR_TRIG	通道 1 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x070	CH1_AL3_CTRL	通道 1 CTRL 寄存器别名
0x074	CH1_AL3_WRITE_ADDR	通道 1 WRITE_ADDR 寄存器别名
0x078	CH1_AL3_TRANS_COUNT	通道 1 TRANS_COUNT 寄存器别名
0x07c	CH1_AL3_READ_ADDR_TRIG	通道 1 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x080	CH2_READ_ADDR	DMA 通道 2 读地址指针
0x084	CH2_WRITE_ADDR	DMA 通道 2 写地址指针
0x088	CH2_TRANS_COUNT	DMA 通道 2 传输计数
0x08c	CH2_CTRL_TRIG	DMA 通道 2 控制与状态寄存器
0x090	CH2_AL1_CTRL	通道 2 CTRL 寄存器别名
0x094	CH2_AL1_READ_ADDR	通道 2 READ_ADDR 寄存器别名

偏移量	名称	说明
0x098	CH2_AL1_WRITE_ADDR	通道 2 WRITE_ADDR 寄存器别名
0x09c	CH2_AL1_TRANS_COUNT_TRIG	通道 2 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0a0	CH2_AL2_CTRL	通道 2 CTRL 寄存器别名
0x0a4	CH2_AL2_TRANS_COUNT	通道 2 的 TRANS_COUNT 寄存器别名
0x0a8	CH2_AL2_READ_ADDR	通道 2 READ_ADDR 寄存器别名
0x0ac	CH2_AL2_WRITE_ADDR_TRIG	通道 2 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0b0	CH2_AL3_CTRL	通道 2 CTRL 寄存器别名
0x0b4	CH2_AL3_WRITE_ADDR	通道 2 WRITE_ADDR 寄存器别名
0x0b8	CH2_AL3_TRANS_COUNT	通道 2 的 TRANS_COUNT 寄存器别名
0x0bc	CH2_AL3_READ_ADDR_TRIG	通道 2 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0c0	CH3_READ_ADDR	DMA 通道 3 读取地址指针
0x0c4	CH3_WRITE_ADDR	DMA 通道 3 写入地址指针
0x0c8	CH3_TRANS_COUNT	DMA 通道 3 传输计数
0xcc	CH3_CTRL_TRIG	DMA 通道 3 控制与状态寄存器
0xd0	CH3_AL1_CTRL	通道 3 的 CTRL 寄存器别名
0xd4	CH3_AL1_READ_ADDR	通道 3 的 READ_ADDR 寄存器别名
0xd8	CH3_AL1_WRITE_ADDR	通道 3 的 WRITE_ADDR 寄存器别名
0xdc	CH3_AL1_TRANS_COUNT_TRIG	通道 3 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0xe0	CH3_AL2_CTRL	通道 3 的 CTRL 寄存器别名
0xe4	CH3_AL2_TRANS_COUNT	通道 3 的 TRANS_COUNT 寄存器别名
0xe8	CH3_AL2_READ_ADDR	通道 3 的 READ_ADDR 寄存器别名
0xec	CH3_AL2_WRITE_ADDR_TRIG	通道 3 的 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0xf0	CH3_AL3_CTRL	通道 3 的 CTRL 寄存器别名
0xf4	CH3_AL3_WRITE_ADDR	通道 3 的 WRITE_ADDR 寄存器别名
0xf8	CH3_AL3_TRANS_COUNT	通道 3 的 TRANS_COUNT 寄存器别名
0xfc	CH3_AL3_READ_ADDR_TRIG	通道 3 的 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x100	CH4_READ_ADDR	DMA 通道 4 读取地址指针
0x104	CH4_WRITE_ADDR	DMA 通道 4 写入地址指针

偏移量	名称	说明
0x108	CH4_TRANS_COUNT	DMA 通道 4 传输计数
0x10c	CH4_CTRL_TRIG	DMA 通道 4 控制与状态寄存器
0x110	CH4_AL1_CTRL	通道 4 的 CTRL 寄存器别名
0x114	CH4_AL1_READ_ADDR	通道 4 的 READ_ADDR 寄存器别名
0x118	CH4_AL1_WRITE_ADDR	通道 4 的 WRITE_ADDR 寄存器别名
0x11c	CH4_AL1_TRANS_COUNT_TRIG	通道 4 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x120	CH4_AL2_CTRL	通道 4 的 CTRL 寄存器别名
0x124	CH4_AL2_TRANS_COUNT	通道 4 的 TRANS_COUNT 寄存器别名
0x128	CH4_AL2_READ_ADDR	通道 4 的 READ_ADDR 寄存器别名
0x12c	CH4_AL2_WRITE_ADDR_TRIG	通道 4 的 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x130	CH4_AL3_CTRL	通道 4 的 CTRL 寄存器别名
0x134	CH4_AL3_WRITE_ADDR	通道 4 的 WRITE_ADDR 寄存器别名
0x138	CH4_AL3_TRANS_COUNT	通道 4 的 TRANS_COUNT 寄存器别名
0x13c	CH4_AL3_READ_ADDR_TRIG	通道 4 的 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x140	CH5_READ_ADDR	DMA 通道 5 读取地址指针
0x144	CH5_WRITE_ADDR	DMA 通道 5 写入地址指针
0x148	CH5_TRANS_COUNT	DMA 通道 5 传输计数
0x14c	CH5_CTRL_TRIG	DMA 通道 5 控制与状态
0x150	CH5_AL1_CTRL	通道 5 CTRL 寄存器别名
0x154	CH5_AL1_READ_ADDR	通道 5 READ_ADDR 寄存器别名
0x158	CH5_AL1_WRITE_ADDR	通道 5 WRITE_ADDR 寄存器别名
0x15c	CH5_AL1_TRANS_COUNT_TRIG	通道 5 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x160	CH5_AL2_CTRL	通道 5 CTRL 寄存器别名
0x164	CH5_AL2_TRANS_COUNT	通道 5 TRANS_COUNT 寄存器别名
0x168	CH5_AL2_READ_ADDR	通道 5 READ_ADDR 寄存器别名
0x16c	CH5_AL2_WRITE_ADDR_TRIG	通道 5 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x170	CH5_AL3_CTRL	通道 5 CTRL 寄存器别名
0x174	CH5_AL3_WRITE_ADDR	通道 5 WRITE_ADDR 寄存器别名
0x178	CH5_AL3_TRANS_COUNT	通道 5 TRANS_COUNT 寄存器别名

偏移量	名称	说明
0x17c	CH5_AL3_READ_ADDR_TRIG	通道5 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x180	CH6_READ_ADDR	DMA通道6读取地址指针
0x184	CH6_WRITE_ADDR	DMA通道6写入地址指针
0x188	CH6_TRANS_COUNT	DMA通道6传输计数
0x18c	CH6_CTRL_TRIG	DMA通道6控制与状态
0x190	CH6_AL1_CTRL	通道6 CTRL寄存器别名
0x194	CH6_AL1_READ_ADDR	通道6 READ_ADDR寄存器别名
0x198	CH6_AL1_WRITE_ADDR	通道6 WRITE_ADDR寄存器别名
0x19c	CH6_AL1_TRANS_COUNT_TRIG	通道6 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1a0	CH6_AL2_CTRL	通道6 CTRL寄存器别名
0x1a4	CH6_AL2_TRANS_COUNT	通道6 TRANS_COUNT寄存器别名
0x1a8	CH6_AL2_READ_ADDR	通道6 READ_ADDR寄存器别名
0x1ac	CH6_AL2_WRITE_ADDR_TRIG	通道6 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1b0	CH6_AL3_CTRL	通道6 CTRL寄存器别名
0x1b4	CH6_AL3_WRITE_ADDR	通道6 WRITE_ADDR寄存器别名
0x1b8	CH6_AL3_TRANS_COUNT	通道6 TRANS_COUNT寄存器别名
0x1bc	CH6_AL3_READ_ADDR_TRIG	通道6 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1c0	CH7_READ_ADDR	DMA通道7读取地址指针
0x1c4	CH7_WRITE_ADDR	DMA通道7写入地址指针
0x1c8	CH7_TRANS_COUNT	DMA通道7传输计数
0x1cc	CH7_CTRL_TRIG	DMA通道7控制与状态
0x1d0	CH7_AL1_CTRL	通道7 CTRL寄存器别名
0x1d4	CH7_AL1_READ_ADDR	通道7 READ_ADDR寄存器别名
0x1d8	CH7_AL1_WRITE_ADDR	通道7 WRITE_ADDR寄存器别名
0x1dc	CH7_AL1_TRANS_COUNT_TRIG	通道7 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1e0	CH7_AL2_CTRL	通道7 CTRL寄存器别名
0x1e4	CH7_AL2_TRANS_COUNT	通道7 TRANS_COUNT寄存器别名
0x1e8	CH7_AL2_READ_ADDR	通道7 READ_ADDR寄存器别名

偏移量	名称	说明
0x1ec	CH7_AL2_WRITE_ADDR_TRIG	通道7 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1f0	CH7_AL3_CTRL	通道7 CTRL寄存器别名
0x1f4	CH7_AL3_WRITE_ADDR	通道7 WRITE_ADDR寄存器别名
0x1f8	CH7_AL3_TRANS_COUNT	通道7 TRANS_COUNT寄存器别名
0x1fc	CH7_AL3_READ_ADDR_TRIG	通道7 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x200	CH8_READ_ADDR	DMA 通道 8 读取地址指针
0x204	CH8_WRITE_ADDR	DMA 通道 8 写入地址指针
0x208	CH8_TRANS_COUNT	DMA 通道 8 传输计数
0x20c	CH8_CTRL_TRIG	DMA 通道 8 控制与状态寄存器
0x210	CH8_AL1_CTRL	通道 8 CTRL 寄存器别名
0x214	CH8_AL1_READ_ADDR	通道 8 READ_ADDR 寄存器别名
0x218	CH8_AL1_WRITE_ADDR	通道 8 WRITE_ADDR 寄存器别名
0x21c	CH8_AL1_TRANS_COUNT_TRIG	通道 8 TRANS_COUNT 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x220	CH8_AL2_CTRL	通道 8 CTRL 寄存器别名
0x224	CH8_AL2_TRANS_COUNT	通道 8 TRANS_COUNT 寄存器别名
0x228	CH8_AL2_READ_ADDR	通道 8 READ_ADDR 寄存器别名
0x22c	CH8_AL2_WRITE_ADDR_TRIG	通道 8 WRITE_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x230	CH8_AL3_CTRL	通道 8 CTRL 寄存器别名
0x234	CH8_AL3_WRITE_ADDR	通道 8 WRITE_ADDR 寄存器别名
0x238	CH8_AL3_TRANS_COUNT	通道 8 TRANS_COUNT 寄存器别名
0x23c	CH8_AL3_READ_ADDR_TRIG	通道 8 READ_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x240	CH9_READ_ADDR	DMA 通道9 读地址指针
0x244	CH9_WRITE_ADDR	DMA 通道9 写地址指针
0x248	CH9_TRANS_COUNT	DMA 通道9 传输计数
0x24c	CH9_CTRL_TRIG	DMA 通道9 控制与状态
0x250	CH9_AL1_CTRL	通道9 CTRL 寄存器的别名
0x254	CH9_AL1_READ_ADDR	通道9 READ_ADDR寄存器别名
0x258	CH9_AL1_WRITE_ADDR	通道9 WRITE_ADDR寄存器别名

偏移量	名称	说明
0x25c	CH9_AL1_TRANS_COUNT_TRIG	通道9 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x260	CH9_AL2_CTRL	通道9 CTRL 寄存器的别名
0x264	CH9_AL2_TRANS_COUNT	通道9 TRANS_COUNT寄存器别名
0x268	CH9_AL2_READ_ADDR	通道9 READ_ADDR寄存器别名
0x26c	CH9_AL2_WRITE_ADDR_TRIG	通道9 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x270	CH9_AL3_CTRL	通道9 CTRL 寄存器的别名
0x274	CH9_AL3_WRITE_ADDR	通道9 WRITE_ADDR寄存器别名
0x278	CH9_AL3_TRANS_COUNT	通道9 TRANS_COUNT寄存器别名
0x27c	CH9_AL3_READ_ADDR_TRIG	通道9 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x280	CH10_READ_ADDR	DMA通道10读地址指针
0x284	CH10_WRITE_ADDR	DMA通道10写地址指针
0x288	CH10_TRANS_COUNT	DMA 通道 10 传输计数
0x28c	CH10_CTRL_TRIG	DMA 通道 10 控制与状态
0x290	CH10_AL1_CTRL	通道 10 CTRL 寄存器别名
0x294	CH10_AL1_READ_ADDR	通道 10 READ_ADDR 寄存器别名
0x298	CH10_AL1_WRITE_ADDR	通道 10 WRITE_ADDR 寄存器别名
0x29c	CH10_AL1_TRANS_COUNT_TRIG	通道 10 TRANS_COUNT 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2a0	CH10_AL2_CTRL	通道 10 CTRL 寄存器别名
0x2a4	CH10_AL2_TRANS_COUNT	通道 10 TRANS_COUNT 寄存器别名
0x2a8	CH10_AL2_READ_ADDR	通道 10 READ_ADDR 寄存器别名
0x2ac	CH10_AL2_WRITE_ADDR_TRIG	通道 10 WRITE_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2b0	CH10_AL3_CTRL	通道 10 CTRL 寄存器别名
0x2b4	CH10_AL3_WRITE_ADDR	通道 10 WRITE_ADDR 寄存器别名
0x2b8	CH10_AL3_TRANS_COUNT	通道 10 TRANS_COUNT 寄存器别名
0x2bc	CH10_AL3_READ_ADDR_TRIG	通道 10 READ_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2c0	CH11_READ_ADDR	DMA通道11读取地址指针
0x2c4	CH11_WRITE_ADDR	DMA通道11写入地址指针
0x2c8	CH11_TRANS_COUNT	DMA通道11传输计数

偏移量	名称	说明
0x2cc	CH11_CTRL_TRIG	DMA通道11控制与状态
0x2d0	CH11_AL1_CTRL	通道11 CTRL 寄存器别名
0x2d4	CH11_AL1_READ_ADDR	通道11 READ_ADDR 寄存器别名
0x2d8	CH11_AL1_WRITE_ADDR	通道11 WRITE_ADDR 寄存器别名
0x2dc	CH11_AL1_TRANS_COUNT_TRIG	通道11 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x2e0	CH11_AL2_CTRL	通道11 CTRL 寄存器别名
0x2e4	CH11_AL2_TRANS_COUNT	通道11 TRANS_COUNT 寄存器别名
0x2e8	CH11_AL2_READ_ADDR	通道11 READ_ADDR 寄存器别名
0x2ec	CH11_AL2_WRITE_ADDR_TRIG	通道11 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x2f0	CH11_AL3_CTRL	通道11 CTRL 寄存器别名
0x2f4	CH11_AL3_WRITE_ADDR	通道11 WRITE_ADDR 寄存器别名
0x2f8	CH11_AL3_TRANS_COUNT	通道11 TRANS_COUNT 寄存器别名
0x2fc	CH11_AL3_READ_ADDR_TRIG	通道11 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x400	INTR	中断状态 (原始)
0x404	INTE0	IRQ 0中断使能
0x408	INTF0	强制中断
0x40c	INTS0	IRQ 0 的中断状态
0x414	INTE1	IRQ 1 的中断使能
0x418	INTF1	IRQ 1 的强制中断
0x41c	INTS1	IRQ 1 的中断状态 (屏蔽)
0x420	TIMER0	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。
0x424	TIMER1	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。
0x428	TIMER2	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。

偏移量	名称	说明
0x42c	TIMER3	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次（即持续 TREQ）或更低速率产生 TREQ。
0x430	MULTI_CHAN_TRIGGER	同时触发一个或多个通道
0x434	SNIFF_CTRL	嗅探器控制
0x438	SNIFF_DATA	嗅探硬件数据累加器
0x440	FIFO_LEVELS	调试 RAF、WAF、TDF 级别
0x444	CHAN_ABORT	中止一个或多个通道上的正在进行传输序列
0x448	N_CHANNELS	该 DMA 实例配置的通道数量。 该 DMA 最多支持 16 个硬件通道，但可配置为最少 1 个，以减少芯片面积。
0x800	CH0_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x804	CH0_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x840	CH1_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x844	CH1_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x880	CH2_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x884	CH2_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x8c0	CH3_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x8c4	CH3_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x900	CH4_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x904	CH4_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度

偏移量	名称	说明
0x940	CH5_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x944	CH5_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x980	CH6_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x984	CH6_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x9c0	CH7_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x9c4	CH7_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa00	CH8_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa04	CH8_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa40	CH9_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa44	CH9_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa80	CH10_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa84	CH10_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xac0	CH11_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xac4	CH11_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度

DMA: CH0_READ_ADDR, CH1_READ_ADDR, ..., CH10_READ_ADDR, CH11_READ_ADDR 寄存器

偏移量: 0x000, 0x040, ..., 0x280, 0x2c0

描述

DMA 通道 N 读地址指针

表 121
 CH0_READ_ADDR,
 CH1_READ_ADDR, ...
 CH10_READ_ADDR,
 CH11_READ_ADDR
 寄存器

位	描述	类型	复位值
31:0	该寄存器在每次读完成时自动更新。当前值为该通道下次待读的地址。	读写	0x00000000

DMA: CH0_WRITE_ADDR, CH1_WRITE_ADDR, ..., CH10_WRITE_ADDR, CH11_WRITE_ADDR 寄存器

偏移量: 0x004, 0x044, ..., 0x284, 0x2c4

描述

DMA 通道 N 写地址指针

表 122。
 CH0_WRITE_ADDR,
 CH1_WRITE_ADDR, ...
 CH10_WRITE_ADDR,
 CH11_WRITE_ADDR
 寄存器

位	描述	类型	复位值
31:0	该寄存器在每次写入完成后自动更新。当前值为该通道将要写入的下一个地址。	读写	0x00000000

DMA: CH0_TRANS_COUNT, CH1_TRANS_COUNT, ..., CH10_TRANS_COUNT, CH11_TRANS_COUNT 寄存器

偏移量: 0x008, 0x048, ..., 0x288, 0x2c8

描述

DMA 通道 N 传输计数

表 123。
 CH0_TRANS_COUNT,
 CH1_TRANS_COUNT,
 ...
 CH10_TRANS_COUNT,
 CH11_TRANS_COUNT
 寄存器

位	描述	类型	复位值
31:0	<p>设置通道在停止前将执行的总线传输次数。</p> <p>注意，若传输大小超过一个字节，该数值不等于传输的字节数（参见CTRL_DATA_SIZE）。</p> <p>当通道处于活动状态时，读取该寄存器显示剩余传输次数，且在每次写传输完成后自动更新。</p> <p>写入此寄存器将设置传输计数器的 RELOAD 值。每次触发该通道时，RELOAD 值将被复制到实时传输计数器。该通道可多次启动，每次启动将执行相同次数的传输，次数由最近一次写入的值确定。</p> <p>RELOAD 值可通过 CHx_DBG_TCR 进行观察。若以 TRANS_COUNT 作为触发条件，写入值将立即用作新传输序列的长度，并写入 RELOAD。</p>	读写	0x00000000

DMA: CH0_CTRL_TRIG, CH1_CTRL_TRIG, ..., CH10_CTRL_TRIG, CH11_CTRL_TRIG 寄存器

偏移量: 0x00c, 0x04c, ..., 0x28c, 0x2cc

描述

DMA 通道 N 控制与状态寄存器

表 124。
 $CH0_CTRL_TRIG$
 $\backslash CH1_CTRL_TRIG$
 $\backslash \dots CH10_CTR$
 $L_TRIG, CH11_C$
 TRL_TRIG 寄存器

位	描述	类型	复位值
31	AHB_ERROR: READ_ERROR 与 WRITE_ERROR 标志的逻辑或。当遇到任意总线错误时，通道将停止，并始终触发其通道 IRQ 标志。	只读	0x0
30	READ_ERROR: 若为1，表示通道接收到读总线错误。写入1可清除该标志。READ_ADDR 显示遇到总线错误的大致地址（不会早于实际发生地址，且不会晚于3次传输后）。	WC	0x0
29	WRITE_ERROR: 若为1，表示通道接收到写总线错误。写入1可清除该标志。WRITE_ADDR 显示遇到总线错误的大致地址（不会早于实际发生地址，且不会晚于5次传输后）。	WC	0x0
28:25	保留。	-	-
24	BUSY: 当通道开始新的传输序列时，该标志置高；当该序列的最后一次传输完成时，标志置低。在 BUSY 标志为高时清除 EN 会暂停通道，且暂停期间 BUSY 标志保持为高。 如需提前终止序列（并清除 BUSY 标志），请参见 CHAN_ABORT。	只读	0x0
23	SNIFF_EN: 若为1，本通道的数据传输对嗅探硬件可见，且每次传输将使校验和状态前进。此项仅在嗅探硬件启用且选择了该通道时适用。 此设置允许基于每个控制块启用或禁用校验和功能。	读写	0x0
22	BSWAP: 对DMA数据应用字节交换变换。 对于字节数据，此设置无效。对于半字数据，每个半字的两个字节将被交换。对于字数据，每个字的四个字节将被交换以实现逆序。	读写	0x0
21	IRQ QUIET: 在静默（QUIET）模式下，该通道在每个传输块结束时不会产生中断请求（IRQ）。相反，当触发寄存器写入NULL时，会产生中断请求，表明控制块链已结束。 在传输包含多个小控制块的DMA链时，该机制减少了CPU需处理的中断次数。	读写	0x0
20:15	TREQ_SEL: 选择传输请求信号。 该通道使用传输请求信号以调整数据传输速率。 传输请求信号的来源包括内部（定时器）或外部（系统发出的数据请求DR EQ）。 0x0 到 0x3a → 选择 DREQ n 作为 TREQ	读写	0x00
	枚举值：		
	0x3b → TIMER0：选择定时器 0 作为 TREQ		
	0x3c → TIMER1：选择定时器 1 作为 TREQ		
	0x3d → TIMER2：选择定时器 2 作为 TREQ（可选）		
	0x3e → TIMER3：选择定时器 3 作为 TREQ（可选）		
	0x3f → PERMANENT：永久请求，适用于无节奏传输。		
14:11	CHAIN_TO: 当此通道完成时，将触发由 CHAIN_TO 指示的通道。通过设置 C HAIN_TO = (本通道) 来禁用。	读写	0x0
10	RING_SEL: 选择 RING_SIZE 应用于读地址还是写地址。 若为 0，则读地址围绕 ($1 \ll RING_SIZE$) 边界回绕。若为 1，则写地址回绕。	读写	0x0

位	描述	类型	复位值
9:6	<p>RING_SIZE: 地址回绕区域的大小。若为 0，则不回绕。对于 $n > 0$ 的值，仅地址的低 n 位会发生变化。该操作将地址限制在 $(1 \ll n)$ 字节的边界内，便于访问自然对齐的环形缓冲区。</p> <p>环形缓冲区的大小范围为 2 至 32768 字节。此功能可根据 RING_SEL 的取值，应用于读地址或写地址。</p>	读写	0x0
	枚举值：		
	0x0 → RING_NONE		
5	<p>INCR_WRITE: 若值为 1，写地址会在每次传输后递增。若值为 0，每次写入均指向相同的初始地址。</p> <p>通常对此功能应禁止，以适用于内存到外设的传输。</p>	读写	0x0
4	<p>INCR_READ: 若值为 1，读地址会在每次传输后递增。若值为 0，每次读取均指向相同的初始地址。</p> <p>通常对此功能应禁止，以适用于外设到内存的传输。</p>	读写	0x0
3:2	<p>DATA_SIZE: 设置每次总线传输的数据大小（字节／半字／字）。READ_ADDR 和 WRITE_ADDR 会在每次传输后，按此大小（1／2／4 字节）递增。</p>	读写	0x0
	枚举值：		
	0x0 → SIZE_BYTE		
	0x1 → SIZE_HALFWORD		
	0x2 → SIZE_WORD		
1	<p>HIGH_PRIORITY: HIGH_PRIORITY 使通道在请求调度中享有优先权：每个调度周期内，所有高优先级通道首先被考虑，然后仅调度一个低优先级通道，之后再返回高优先级通道。</p> <p>此设置仅影响 DMA 调度通道的顺序。DMA 的总线优先级保持不变。若 DMA 未达到饱和状态，低优先级通道的吞吐量不会受到影响。</p>	读写	0x0
0	<p>EN: DMA 通道使能。</p> <p>当该位为 1 时，通道响应触发事件，导致其进入 BUSY 状态并开始传输数据。当该位为 0 时，通道忽略触发，停止发出传输请求，并暂停当前传输序列（即若 BUSY 已处于高电平则保持该状态）。</p>	读写	0x0

DMA: CH0_AL1_CTRL, CH1_AL1_CTRL, ..., CH10_AL1_CTRL, CH11_AL1_CTRL 寄存器

偏移量: 0x010, 0x050, ..., 0x290, 0x2d0

表125。
CH0_AL1_CTRL
、 CH1_AL1_CTRL
、 ...、 CH10_AL1
_CTRL、 CH11_A
L1_CTRL寄存器

位	描述	类型	复位值
31:0	通道 NCTRL寄存器别名	读写	-

DMA: CH0_AL1_READ_ADDR, CH1_AL1_READ_ADDR, ..., CH10_AL1_READ_ADDR、 CH11_AL1_READ_ADDR寄存器

偏移量: 0x014, 0x054, ..., 0x294, 0x2d4

表126。
CH0_AL1_READ_ADDR
, CH1_AL1_READ_ADDR
,...
CH10_AL1_READ_ADDR
R,
CH11_AL1_READ_ADDR
R寄存器

位	描述	类型	复位值
31:0	通道 NREAD_ADDR寄存器别名	读写	-

DMA: CH0_AL1_WRITE_ADDR, CH1_AL1_WRITE_ADDR, ..., CH10_AL1_WRITE_ADDR、 CH11_AL1_WRITE_ADDR寄存器

偏移量: 0x018, 0x058, ..., 0x298, 0x2d8

表127。
CH0_AL1_WRITE_ADD
R,
CH1_AL1_WRITE_ADD
R,...
CH10_AL1_WRITE_ADD
DR,
CH11_AL1_WRITE_ADD
DR寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR寄存器别名	读写	-

DMA: CH0_AL1_TRANS_COUNT_TRIG, CH1_AL1_TRANS_COUNT_TRIG, ..., CH10_AL1_TRANS_COUNT_TRIG, CH11_AL1_TRANS_COUNT_TRIG寄存器

偏移量: 0x01c, 0x05c, ..., 0x29c, 0x2dc

表128。
CH0_AL1_TRANS_CO
NT_TRIG,
CH1_AL1_TRANS_CO
NT_TRIG, ...
CH10_AL1_TRANS_CO
UNT_TRIG,
CH11_AL1_TRANS_CO
UNT_TRIG寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: CH0_AL2_CTRL, CH1_AL2_CTRL, ..., CH10_AL2_CTRL, CH11_AL2_CTRL寄存器

偏移: 0x020, 0x060, ..., 0x2a0, 0x2e0

表129。
CH0_AL2_CTRL
、 CH1_AL2_CTRL
、 ...、 CH10_AL2_CT
RL、 CH11_AL2_CT
寄存器

位	描述	类型	复位值
31:0	通道 NCTRL寄存器别名	读写	-

DMA: CH0_AL2_TRANS_COUNT、 CH1_AL2_TRANS_COUNT、 ...、 CH10_AL2_TRANS_COUNT、 CH11_AL2_TRANS_COUNT 寄存器

偏移: 0x024, 0x064, ..., 0x2a4, 0x2e4

表130。
CH0_AL2_TRANS_CO
UNT、 CH1_AL2
_TRANS_COUNT、 ...
、 CH10_AL2_T
RANS_COUNT、 CH11
_AL2_TRANS_C
OUNT寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT寄存器别名	读写	-

DMA: CH0_AL2_READ_ADDR, CH1_AL2_READ_ADDR, ..., CH10_AL2_READ_ADDR、 CH11_AL2_READ_ADDR 寄存器

偏移: 0x028, 0x068, ..., 0x2a8, 0x2e8

表131。
CH0_AL2_READ_ADDR
,
CH1_AL2_READ_ADDR
,

位	描述	类型	复位值
31:0	通道 NREAD_ADDR 寄存器别名	读写	-

DMA: CH0_AL2_WRITE_ADDR_TRIG, CH1_AL2_WRITE_ADDR_TRIG, ..., CH10_AL2_WRITE_ADDR_TRIG, CH11_AL2_WRITE_ADDR_TRIG 寄存器

偏移地址: 0x02c, 0x06c, ..., 0x2ac, 0x2ec

表132。
CH0_AL2_WRITE_ADD_R_TRIG,
CH1_AL2_WRITE_ADD_R_TRIG, ...
CH10_AL2_WRITE_AD_DR_TRIG,
CH11_AL2_WRITE_DR_TRIG 寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: CH0_AL3_CTRL, CH1_AL3_CTRL, ..., CH10_AL3_CTRL, CH11_AL3_CTRL 寄存器

偏移地址: 0x030, 0x070, ..., 0x2b0, 0x2f0

表133。
CH0_AL3_CTRL,
CH1_AL3_CTRL, ...
, CH10_AL3_CTRL
, CH11_AL3_CTRL
L 寄存器

位	描述	类型	复位值
31:0	通道 NCTRL 寄存器别名	读写	-

DMA: CH0_AL3_WRITE_ADDR, CH1_AL3_WRITE_ADDR, ..., CH10_AL3_WRITE_ADDR, CH11_AL3_WRITE_ADDR 寄存器

偏移量: 0x034, 0x074, ..., 0x2b4, 0x2f4

表134。
CH0_AL3_WRITE_ADD_R,
CH1_AL3_WRITE_ADD_R, ...
CH10_AL3_WRITE_AD_DR,
CH11_AL3_WRITE_DR 寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR 寄存器别名	读写	-

DMA: CH0_AL3_TRANS_COUNT, CH1_AL3_TRANS_COUNT, ..., CH10_AL3_TRANS_COUNT, CH11_AL3_TRANS_COUNT 寄存器

偏移量: 0x038, 0x078, ..., 0x2b8, 0x2f8

表135。
CH0_AL3_TRANS_CO_NT,
CH1_AL3_TRANS_CO_NT, ...
CH10_AL3_TRANS_CO_NT,
CH11_AL3_TRANS_CO_NT 寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT 寄存器别名	读写	-

DMA: CH0_AL3_READ_ADDR_TRIG, CH1_AL3_READ_ADDR_TRIG, ..., CH10_AL3_READ_ADDR_TRIG, CH11_AL3_READ_ADDR_TRIG 寄存器

偏移量: 0x03c, 0x07c, ..., 0x2bc, 0x2fc

表136。
CH0_AL3_READ_ADDR_TRIG,
CH1_AL3_READ_ADDR_TRIG, ...
CH10_AL3_READ_ADD_R_TRIG,
CH11_AL3_READ_DR_TRIG 寄存器

位	描述	类型	复位值
31:0	通道 NREAD_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: INTR 寄存器

偏移量: 0x400

描述

中断状态 (原始)

表137。INTR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	DMA 通道 0 至 15 的原始中断状态。位 n 对应通道 n。 忽略任何屏蔽或强制操作。可通过向 INTR、INTS0 或 INTS1 写入位掩码以清除通道中断。 根据 INTE0 和 INTE1，通道中断可路由至两个系统级 IRQ 的任一。 此功能可用于将不同通道中断向量分配至不同中断服务程序（ISR）：此举可能旨在允许对更具时效性的通道执行 NVIC IRQ 抢占，或将 IRQ 负载分散至不同核心。 同样可忽略该行为，仅使用 INTE0/INTS0/IRQ 0。	WC	0x0000

DMA：INTE0 寄存器

偏移: 0x404

描述

IRQ 0 中断使能

表 138。INTE0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	设置位 n 以使通道 n 的中断传递至 DMA IRQ 0。	读写	0x0000

DMA：INTF0 寄存器

偏移: 0x408

描述

强制中断

表 139。INTF0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	写入 1 以强制 INTE0 中相应位的中断。中断将持续有效，直到 INTF0 被清除。	读写	0x0000

DMA：INTS0 寄存器

偏移: 0x40c

描述

IRQ 0 的中断状态

表140. INTS0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	指示当前激活的通道中断请求，这些请求正导致 IRQ 0 被触发。 可通过向此处写入位掩码以清除通道中断。	WC	0x0000

DMA：INTE1 寄存器

偏移: 0x414

描述

IRQ 1 的中断使能

表141. INTE1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	设置第 n 位以将第 n 通道的中断传递至 DMA IRQ 1。	读写	0x0000

DMA：INTF1 寄存器

偏移: 0x418

描述

IRQ 1 的强制中断

表142. INTF1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	写入 1 以强制 INTFO 中相应位的中断。中断将持续有效，直到 INTFO 被清除。	读写	0x0000

DMA：INTS1 寄存器

偏移: 0x41c

描述

IRQ 1 的中断状态（屏蔽）

表143. INTS1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	指示当前激活的通道中断请求，这些请求正导致 IRQ 1 被触发。 可通过向此处写入位掩码以清除通道中断。	WC	0x0000

DMA：TIMER0、TIMER1、TIMER2、TIMER3 寄存器

偏移: 0x420, 0x424, 0x428, 0x42c

描述

节拍 (X/Y) 分数定时器

该节拍定时器以 $((X/Y) * \text{sys_clk})$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次（即持续 TREQ）或更低速率产生 TREQ。

表 144。 TIMER0,
TIMER1, TIMER2,
TIMER3 寄存器

位	描述	类型	复位值
31:16	X：分频定时器被除数。指定 (X/Y) 分数定时器的 X 值。	读写	0x0000

位	描述	类型	复位值
15:0	Y：分频定时器除数。指定 (X/Y) 分数定时器的 Y 值。	读写	0x0000

DMA: MULTI_CHAN_TRIGGER 寄存器

偏移: 0x430

描述

同时触发一个或多个通道

表 145。
MULTI_CHAN_TRIGGER
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	该寄存器的每个位对应一个 DMA 通道。向相关位写入 1 等同于写入该通道的触发寄存器；若通道已启用且当前未忙，则将启动该通道。	SC	0x0000

DMA: SNIFF_CTRL 寄存器

偏移: 0x434

描述

嗅探器控制

表 146。
SNIFF_CTRL 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	OUT_INV : 若设置，读取时结果将呈现取反（按位补码）。此设置不影响校验和的计算方式；结果在结果寄存器与总线之间实时转换。	读写	0x0
10	OUT_REV : 若设置，读取时结果按位反转显示。此设置不影响校验和的计算方式；结果在结果寄存器与总线之间实时转换。	读写	0x0
9	BSWAP : 在输入校验和前，对嗅探数据局部执行字节反转。 请注意，嗅探硬件位于 DMA 通道字节交换操作之后的下游：若通道 CTRL_BSWAP 与 SNIFF_CTRL_BSWAP 均启用，则从嗅探器视角两者效果相互抵消。	读写	0x0
8:5	CALC 枚举值： 0x0 → CRC32: 计算 CRC-32 (IEEE802.3 多项式)	读写	0x0
	0x1 → CRC32R: 计算位反转数据的 CRC-32 (IEEE802.3 多项式)		
	0x2 → CRC16: 计算 CRC-16-CCITT		
	0x3 → CRC16R: 计算位反转数据的 CRC-16-CCITT		
	0xe → EVEN: 对所有数据执行 XOR 归约。== 1 若总计1的人口数为奇数。		
	0xf → SUM: 计算简单的32位校验和（使用32位累加器相加）		

位	描述	类型	复位值
4:1	DMACH : 供嗅探器观察的DMA通道	读写	0x0
0	EN : 启用嗅探器	读写	0x0

DMA: SNIFF_DATA 寄存器

偏移量: 0x438

说明

嗅探硬件数据累加器

表147
SNIFF_DATA 寄存器

位	描述	类型	复位值
31:0	在开始对由 SNIFF_CTRL_DMACH 指示的通道进行DMA传输前, 请在此写入初始种子值。硬件将在每次侦测到对该通道的读取时更新此寄存器。通道完成后, 可从该寄存器读取最终结果。	读写	0x00000000

DMA: FIFO_LEVELS 寄存器

偏移量: 0x440

说明

调试 RAF、WAF、TDF 级别

表148
FIFO_LEVELS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	RAF_LVL : 当前读地址FIFO填充级别	只读	0x00
15:8	WAF_LVL : 当前写地址FIFO填充级别	只读	0x00
7:0	TDF_LVL : 当前传输数据FIFO填充级别	只读	0x00

DMA: CHAN_ABORT 寄存器

偏移: 0x444

描述

中止一个或多个通道上的正在进行传输序列

表149。
CHAN_ABORT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	每个位对应一个通道。写入1会中止该通道上正在进行的任何传输序列。该位将保持高电平, 直到所有在途传输通过地址和数据FIFO完全刷新。 写入后, 必须轮询该寄存器, 直到其返回全零。在此之前, 重启通道是不安全的。	SC	0x0000

DMA: N_CHANNELS 寄存器

偏移: 0x448

表150。
N_CHANNELS 寄存器

位	描述	类型	复位值
31:5	保留。	-	-

位	描述	类型	复位值
4:0	本DMA实例所配置的通道数量。该DMA最多支持16个硬件通道，但可配置为最少1个，以减少芯片面积。	只读	-

DMA: CH0_DBG_CTDREQ, CH1_DBG_CTDREQ, ..., CH10_DBG_CTDREQ, CH11_DBG_CTDREQ 寄存器

偏移: 0x800, 0x840, ..., 0xa80, 0xac0

表151。
CH0_DBG_CTDREQ,
, CH1_DBG_CTDREQ
, ..., CH10_DBG_C
TDREQ, CH11_DB
G_CTDREQ 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	读取: 获取通道DREQ计数器(即DMA预计可对外设执行而不会溢出或欠载的访问次数)。写入任意值: 清除计数器，并使通道重新启动DREQ握手。	WC	0x00

DMA: CH0_DBG_TCR, CH1_DBG_TCR, ..., CH10_DBG_TCR, CH11_DBG_TCR 寄存器

偏移量: 0x804, 0x844, ..., 0xa84, 0xac4

表152。
CH0_DBG_TCR,
CH1_DBG_TCR, ...
CH10_DBG_TCR,
CH11_DBG_TCR
寄存器

位	描述	类型	复位值
31:0	读取通道TRANS_COUNT重新加载值，即下一次传输的长度	只读	0x00000000

2.6. 内存

RP2040内置ROM和SRAM，并通过QSPI接口访问外部Flash。内部存储器详细信息如下。

2.6.1. 只读存储器 (ROM)

一块16kB只读存储器(ROM)位于地址 0x00000000. ROM内容在硅片制造时固定，包含：

- 初始启动程序
- Flash启动序列
- Flash编程例程
- 支持UF2的USB大容量存储设备
- 快速浮点等实用库

芯片的启动序列定义于第2.8.1节，ROM内容在第2.8节中有更详细的描述。RP2040启动ROM的完整源代码可在以下位置获取：

[pico-bootrom](#)

ROM提供单周期只读总线访问，且位于专用的AHB-Lite仲裁器上，因此可与其他存储设备同时访问。尝试写入ROM不会产生任何效果(不会触发总线错误)。

2.6.2. 静态随机存取存储器 (SRAM)

芯片内共有 264kB SRAM。物理上划分为六个存储块，这显著提升了多主设备的内存带宽，但软件可将其视为单一 264kB 内存区域。对每个存储块中的存储内容无限制：可为处理器代码、数据缓冲区或二者混合。共有四个16k x 32位存储区（每个64kB）和两个1k x 32位存储区（每个4kB）。

重要

分区是一种物理划分SRAM的方法，通过允许多个同时访问来提升性能。逻辑上存在一个单一的264kB连续内存空间。

每个SRAM存储区均通过专用的AHB-Lite仲裁器进行访问。这意味着不同的总线主控可以并行访问不同的SRAM存储区，因此每个系统时钟周期最多可执行四次32位SRAM访问（每个主控一次）。

SRAM映射至系统地址，起始地址为 `0x20000000`。首个256kB地址区域采用字条带方式跨越四个较大存储区分布，为大多数应用提供显著的内存并行优势。

系统地址空间中连续的字按照表153所示路由至不同的RAM存储区。

表153。SRAM
存储区0/1/2/3条带
映射示意。

系统地址	SRAM银行	SRAM字地址
<code>0x20000000</code>	银行0	0
<code>0x20000004</code>	银行1	0
<code>0x20000008</code>	银行2	0
<code>0x2000000c</code>	银行3	0
<code>0x20000010</code>	银行0	1
<code>0x20000014</code>	银行1	1
<code>0x20000018</code>	银行2	1
<code>0x2000001c</code>	银行3	1
<code>0x20000020</code>	银行0	2
<code>0x20000024</code>	银行1	2
<code>0x20000028</code>	银行2	2
<code>0x2000002c</code>	银行3	2
等等		

接下来的两个4kB区域（起始地址为 `0x20040000` 和 `0x20041000`）直接映射到较小的4kB内存块。

软件可以选择将这些用于每核用途，例如堆栈和频繁执行的代码，以确保处理器访问这些区域时不会发生停滞。然而，与 RP2040 上所有 SRAM 一样，这些内存块对所有主控器提供单周期访问，前提是同一周期内没有其他主控器访问该内存块，因此将内存视为单一的264kB设备是合理的。

这四个64kB的内存块也可通过非条带化镜像访问。从 `0x21000000`、`0x21010000`、`0x21020000`、`0x21030000`起始的四个64kB区域各自直接映射到四个64kB SRAM内存块之一。软件可以明确分配数据和代码至物理内存块，以在极端要求情况下提升内存性能。这通常是不必要的，因为内存条带通常能够提供足够的并行性，并且软件复杂度更低。

非条带镜像起始于SRAM基址上方偏移+16MB处，这是允许ARMv6M在较小存储区与非条带较大存储区之间进行子程序调用的最大偏移。

2.6.2.1. 其他片上存储器

除了264kB主存储器外，还有两个在某些情况下可用的专用RAM块：

- 如果禁用闪存XIP缓存，该缓存将作为一个16kB的存储器，可从 `0x15000000` 开始使用。
- 如果未使用USB，USB数据DPRAM可作为一个4kB存储器，从 `0x50100000` 开始使用。

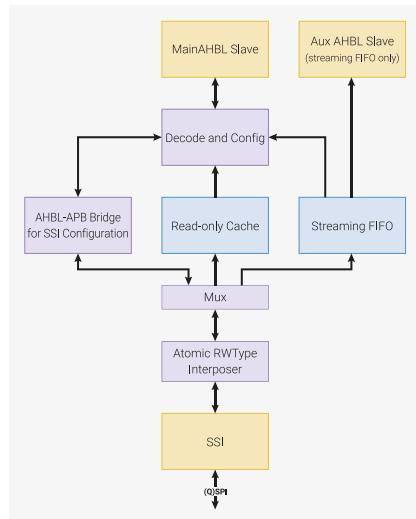
这样，片上SRAM总容量达到284kB。对于这些存储器的使用没有限制，例如，可以选择从USB数据RAM执行代码。

2.6.3. 闪存

外部闪存通过QSPI接口使用执行就地（XIP）硬件进行访问。这使得系统能够将外部闪存作为内部内存一样进行寻址和访问。对起始地址为 `0x1` 的16MB内存窗口的总线读取 `00000000` 会被转换为串行闪存传输，结果返回给发起读取的主设备。该过程对主设备透明，因此处理器可以直接从外部闪存执行代码，无需先将代码复制到内部内存，故称“执行就地”。内部缓存会记录最近访问的闪存内容，从而提升接口的平均带宽和延迟表现。

一旦由RP2040引导ROM及闪存第二阶段正确配置，XIP硬件基本透明，软件可将闪存视为大型只读存储器。然而，它确实提供多项附加功能，以满足更复杂的软件应用需求。

图14。Flash
执行就地（XIP）子
系统。系统通过
主 AHB-Lite 从属设
备的访问会被解
码，以确认访问属
于 XIP 访问
、对 SSI 的直
接访问（例如用
于配置），或对 XI
P 子系统中其他各
种硬件与控制寄
存器的访问。XIP 访
问首先在缓存中
查询，以加速
对最近使用数
据的访问。若缓
存中无数据，则通
过 SSI 发起外部串
行访问，随后
将获取的数据存
入缓存，并
转发至系统总线。
转自：https://www.raspberrypi.org/documentation/hardware/rpi4/technical/flash-xip-subsystem.pdf



注意

使用SDK时，闪存第二阶段将串行闪存接口配置为系统时钟的整数分频。所有包含的第二阶段引导实现均支持 `PICO_FLASH_SPI_CLKDIV` 设置（例如，在 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/boot_stage2/boot2_w25q080.S 中默认值为 4，使默认接口速度为 $125/4 = 31.25\text{MHz}$ ）。该分频设置可通过在 SDK 所用特定板配置头文件中指定 `PICO_FLASH_SPI_CLKDIV` 进行覆盖。

2.6.3.1. XIP 缓存

缓存容量为16kB，双路组相联，命中时间为1周期。它属于XIP子系统内部，仅影响对XIP闪存的访问，因此软件无需考虑缓存一致性，除非执行闪存编程操作。它缓存来自24位闪存地址空间的读取，该地址空间在RP2040地址空间中被多次镜像，每个镜像具有不同的缓存行为。系统地址的最高八位用于段解码，剩余24位用于闪存寻址，因此XIP操作支持的最大闪存容量为16MB。可用的镜像地址

为：

- **0x10…** XIP访问，可缓存，分配——正常缓存操作
- **0x11…** XIP访问，可缓存，不分配——检查命中，未命中时不更新缓存
- **0x12…** XIP访问，不缓存，分配——不检查命中，始终更新缓存
- **0x13…** XIP访问，非缓存，非分配——完全绕过缓存
- **0x15…** 使用XIP缓存作为SRAM组，镜像映射至整个段

如果通过CTRL.EN寄存器位禁用缓存，则所有四个XIP别名（**0x10**至**0x13**）将绕过缓存，直接访问闪存。这对XIP代码执行性能具有显著影响。

除非通过清除CTRL.EN禁用缓存，否则访问**0x15…**段会产生总线错误。一旦禁用缓存，该区域表现为额外的16KB SRAM组。读写操作均为一个周期，但在连续写-读序列中存在等待周期，即不存在写转发缓冲。

2.6.3.2. 缓存刷新与维护

FLUSH寄存器允许刷新整个缓存内容。如果软件重新编程了闪存内容，需要在不重启的情况下清除过期数据和代码，此操作是必要的。缓存刷新可通过向FLUSH写入1手动触发，或在将XIP块从复位状态切出时自动触发。刷新操作通过使用内部计数器将缓存标签存储器清零实现，耗时略超过1024个时钟周期（16kB总容量 / 每行8字节 / 每组2路）。

在访问闪存数据时执行缓存刷新（例如一核发起刷新，另一核可能正在执行闪存代码）是安全的，但任何在刷新进行期间访问闪存数据的主控设备将被阻塞，直至刷新完成。

⚠ 注意

缓存刷新进行时，禁止写入缓存映射的SRAM别名区域（**0x15…**）。首次写入前，如近期已启动缓存刷新（例如通过看门狗复位），建议对FLUSH执行一次虚假读取，以确保缓存刷新完成。刷新进行时写入缓存映射为SRAM可能会破坏数据存储内容。

完全缓存刷新会显著降低后续代码的执行速度，直至缓存重新“预热”。存在另一种方法，允许仅使对应于特定地址范围的缓存内容失效。对**0x10…**镜像的写操作将查找缓存中的对应地址位置，并删除任何匹配的条目。因此，对地址范围内所有字对齐位置的写入操作（例如刚擦除并重新编程的闪存扇区）可消除该范围内陈旧的缓存数据，且不会受到完全缓存刷新带来的影响。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/cache_perfctr/flash_cache_perfctr.c 第30至55行

```

30     // 刷新缓存，确保首次访问 test_data 时缓存失效
31     xip_ctrl_hw->flush = 1;
32     while (!(xip_ctrl_hw->stat & XIP_STAT_FLUSH_READY_BITS))
33         tight_loop_contents();
34
35     // 清除计数器（写入任意值以清零）
36     xip_ctrl_hw->ctr_acc = 1;
37     xip_ctrl_hw->ctr_hit = 1;
38
39     (void) *test_data_ptr;
40     check(xip_ctrl_hw->ctr_hit == 0 && xip_ctrl_hw->ctr_acc == 1,
41           "首次访问数据应未命中");
42
43     (void) *test_data_ptr;
44     check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 2,

```

```

45     "第二次访问数据命中");
46
47 // 写入以失效单个缓存行 (64 位)
48 // 写入必须指向可缓存且可分配的别名 (地址范围0x10.....)
49 *test_data_ptr = 0;
50 (void) *test_data_ptr;
51 check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 3,
52       "失效后应当丢失");
53 (void) *test_data_ptr;
54 check(xip_ctrl_hw->ctr_hit == 2 && xip_ctrl_hw->ctr_acc == 4,
55       "失效后第二次访问应再次命中");

```

2.6.3.3. SSI

执行就地 (execute-in-place) 功能由SSI接口提供，详见第4.10节。该接口支持1位、2位或4位SPI闪存接口（SPI、DSPI及QSPI），并可在每次XIP访问时插入指令前缀或模式续定位。这包括在每次访问时发出标准的 03h串行闪存读命令，几乎可支持任何串行闪存设备。最大SPI时钟频率为系统时钟频率的一半。

SSI还可作为标准的基于FIFO的SPI主控设备使用，并支持DMA。此模式被bootrom用于从外部闪存提取二级引导加载程序（见第2.8.1节）。总线中介器允许对SSI控制寄存器以原子方式执行置位、清除或异或操作，方式与RP2040上的其他内存映射IO相同。此操作详述于第2.1.2节。

2.6.3.4. 闪存流传输与辅助总线从属设备

由于闪存容量通常远大于SRAM，将数据块从闪存流式传输到内存中通常非常有用。在前台软件执行其他任务的同时，让DMA在后台流式传输这些数据，非常便利，更便利的是，在此过程中代码可以继续从闪存执行。

这与标准的XIP操作不兼容，因为在SSI执行串行传输时，DMA会被迫经历长时间的总线阻塞。这些阻塞对处理器而言是可容忍的，因为顺序处理器在等待指令取出完成期间通常无更优任务可做，且典型代码执行的缓存命中率显著高于对不常访问数据的大块流式传输。相反，阻塞DMA会阻止任何其他活动的DMA通道在此期间取得进展，从而降低整体DMA吞吐性能。

STREAM_ADDR和**STREAM_CTR**寄存器用于编程一系列线性闪存读取操作，XIP子系统将以尽最大努力的方式在后台执行这些操作。为了在流媒体持续传输期间最大限度地减少对从闪存执行的代码的影响，流媒体硬件对SSI的访问优先级低于常规的XIP访问，并且在最后一次XIP缓存未命中与恢复流传输之间存在短暂的冷却时间（七个周期）。此举有助于避免在XIP缓存未命中时初始访问延迟的增加。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c 第45至48行

```

45     while (!(xip_ctrl_hw->stat & XIP_STAT_FIFO_EMPTY))
46         (void) xip_ctrl_hw->stream_fifo;
47     xip_ctrl_hw->stream_addr = (uint32_t) &random_test_data[0];
48     xip_ctrl_hw->stream_ctr = count_of(random_test_data);

```

流式数据被推送至一个小型FIFO，该FIFO生成DREQ信号，以告知DMA收集流式数据。由于DMA只有在从闪存读取数据之后才会发起读取请求，因此在访问数据时，DMA不会发生阻塞。

尽管该方案确保一旦DREQ断言，数据即已准备好并位于流FIFO中，但如果其他主设备因XIP从设备阻塞（例如缓存未命中）而暂时停滞，DMA仍可能被阻塞。此问题由辅助总线从设备解决，该从设备为简单总线接口，仅提供对流FIFO的访问。此从设备公开于

FASTPERI仲裁器，该仲裁器仅为不产生等待状态的本地AHB-Lite外设提供服务，因此在假定DMA拥有较高总线优先级的情况下，访问该地址处的FIFO时，DMA将不会发生阻塞。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c 第58至70行

```

58     const uint dma_chan = 0;
59     dma_channel_config cfg = dma_channel_get_default_config(dma_chan);
60     channel_config_set_read_increment(&cfg, false);
61     channel_config_set_write_increment(&cfg, true);
62     channel_config_set_dreq(&cfg, DREQ_XIP_STREAM);
63     dma_channel_configure(
64         dma_chan,
65         &cfg,
66         (void *) buf,           // 写入地址
67         (const void *) XIP_AUX_BASE, // 读取地址
68         count_of(random_test_data), // 传输计数
69         true                   // 立即启动!
70     );

```

2.6.3.5. 性能计数器

XIP子系统提供两个性能计数器。两个计数器均为32位，达到饱和值 `0xffffffff` 时饱和，写入任意值可清零。计数项包括：

1. 任意别名的XIP访问总数
2. 因缓存命中而产生的XIP访问次数

对于常见使用场景，此功能允许对缓存命中率进行分析。

2.6.3.6. XIP 寄存器清单

XIP寄存器起始地址为 `0x14000000`（在SDK中定义为 `XIP_CTRL_BASE`）。

表 154. XIP
寄存器清单

偏移量	名称	说明
0x00	<code>CTRL</code>	缓存控制
0x04	<code>刷新</code>	缓存刷新控制
0x08	<code>状态</code>	缓存状态
0x0c	<code>CTR_HIT</code>	缓存命中计数器
0x10	<code>CTR_ACC</code>	缓存访问计数器
0x14	<code>STREAM_ADDR</code>	FIFO流地址
0x18	<code>STREAM_CTR</code>	FIFO流控制
0x1c	<code>STREAM_FIFO</code>	FIFO流数据

XIP：CTRL 寄存器

偏移：0x00

描述

缓存控制

表 155. CTRL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	POWER_DOWN : 当该位为1时，缓存内存断电，但状态保持，但无法访问。这降低了静态功耗。 向此位写入1将使 CTRL_EN 强制为0，即断电时缓存无法启用。 缓存断电时，作为SRAM使用的缓存访问将触发总线错误响应。	读写	0x0
2	保留。	-	-
1	ERR_BADWRITE : 当值为1时，对除0x0（缓存、分配）外的任何别名的写入将产生总线错误。当值为0时，此类写入将被静默忽略。 在任何情况下，对0x0别名的写入在标签匹配时均按惯例进行取消分配。	读写	0x1
0	EN : 当值为1时，启用缓存。缓存禁用时，所有 XIP 访问将直接访问闪存，不查询缓存。启用缓存后，可缓存的 XIP 访问将查询缓存；当标签匹配且有效位被设置时，闪存不会被访问。 缓存启用时，作为SRAM的缓存访问不会影响缓存数据RAM，且会触发总线错误响应。	读写	0x1

XIP: FLUSH寄存器

偏移量: 0x04

描述

缓存刷新控制

表 156. FLUSH
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	写入1以执行缓存刷新。此操作会清除标签存储器，但数据存储器内容保持不变。（这意味着作为SRAM使用的缓存内容不会受到刷新或复位的影响。） 读取操作将占用总线（阻塞处理器），直至刷新完成。或者可轮询STAT寄存器，直至刷新完成。	SC	0x0

XIP: STAT寄存器

偏移量: 0x08

描述

缓存状态

表 157. STAT
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	FIFO_FULL : 该位为1时，表示XIP流FIFO已满。 流FIFO深度为2，因此满标志和空标志可用以判断FIFO中的数据量。	只读	0x0
1	FIFO_EMPTY : 该位为1时，表示XIP流FIFO为空。	只读	0x1

位	描述	类型	复位值
0	FLUSH_READY : 缓存刷新进行时该位为0，完成后为1。 每当XIP块重置时，缓存即被刷新，亦可通过FLUSH寄存器请求刷新。	只读	0x0

XIP: CTR_HIT寄存器

偏移: 0x0c

描述

缓存命中计数器

表158. CTR_HIT
寄存器

位	描述	类型	复位值
31:0	32位饱和计数器，每当缓存命中时递增，即XIP访问直接由缓存数据提供服务时。 写入任意值可清零。	WC	0x00000000

XIP: CTR_ACC寄存器

偏移: 0x10

描述

缓存访问计数器

表159. CTR_ACC
寄存器

位	描述	类型	复位值
31:0	32位饱和计数器，每当发生XIP访问时递增，无论缓存是否命中，包含非缓存访问。 写入任意值可清零。	WC	0x00000000

XIP: STREAM_ADDR寄存器

偏移: 0x14

描述

FIFO流地址

表160.
STREAM_ADDR
寄存器

位	描述	类型	复位值
31:2	下一字的地址，将从闪存流式传输至FIFO。 每次闪存访问后自动递增。 在开始流式读取前，需将初始访问地址写入此处。	读写	0x00000000
1:0	保留。	-	-

XIP: STREAM_CTR 寄存器

偏移量: 0x18

描述

FIFO 流控制

表 161.
STREAM_CTR 寄存器

位	描述	类型	复位值
31:22	保留。	-	-

位	描述	类型	复位值
21:0	<p>写入非零值以启动流式读取。随后该过程将在后台进行，利用闪存空闲周期将线性数据块从闪存传输至流式 FIFO。</p> <p>随着数据流进展自动递减（每次递减 1），并在达到 0 时停止。</p> <p>写入 0 以停止正在进行的流，并丢弃所有在途读取，以便立即启动新流（在排空 FIFO 并重新初始化 STREAM_ADDR 后）</p> <ul style="list-style-type: none"> ◦ 	读写	0x000000

XIP: STREAM_FIFO 寄存器

偏移量: 0x1c

描述

FIFO 流数据

表 162.
STREAM_FIFO
寄存器

位	描述	类型	复位值
31:0	<p>流式数据在此缓冲，供系统 DMA 读取。</p> <p>该 FIFO 也可通过 XIP_AUX 从属访问，避免 DMA 因其他 XIP 访问导致的总线阻塞。</p>	RF	0x00000000

2.7. 启动顺序

RP2040 的多个组件协同工作，使处理器脱离复位状态并能执行 bootrom（第 2.8 节）。bootrom 是内置芯片的软件，负责执行“处理器控制”的启动序列部分。我们将处理器启动前的步骤称为“硬件控制”启动序列。

硬件控制启动序列如下：

- 电源施加至芯片，RUN 引脚为高电平。（如 RUN 为低电平，芯片将保持复位状态。）
- 片上电压调节器（第 2.10 节）等待数字核心电源（DVDD）稳定。
- 上电状态机（第 2.13 节）启动。序列总结如下：
 - 环形振荡器（第 2.17 节）启动，为时钟发生器提供时钟源。`clk_sys` 和 `clk_ref` 现以相对较低频率（通常为 6.5MHz）运行。
 - 复位控制器（第 2.14 节）、原地执行硬件（第 2.6.3 节）、存储器（第 2.6.2 节和第 2.6.1 节）、总线结构（第 2.1 节）及处理器子系统（第 2.3 节）已退出复位状态。
 - 处理器核心 0 和核心 1 开始执行 Bootrom（第 2.8 节）。

2.8. 启动只读存储器 (Bootrom)

Bootrom 的大小限制为 16kB。其内容包括：

- 处理器核心 0 的初始启动序列。
- 处理器核心 1 的低功耗等待与启动协议。
- 支持 UF2 的 USB MSC 类兼容引导加载程序，用于向 FLASH 或 RAM 下载代码和数据。
- 用于高级管理的 USB PICOBOOT 引导加载程序接口。

- 用于编程和操作外部闪存的例程。
- 快速浮点库。
- 快速位计数与操作函数。
- 快速内存填充与复制函数。

Bootrom源代码

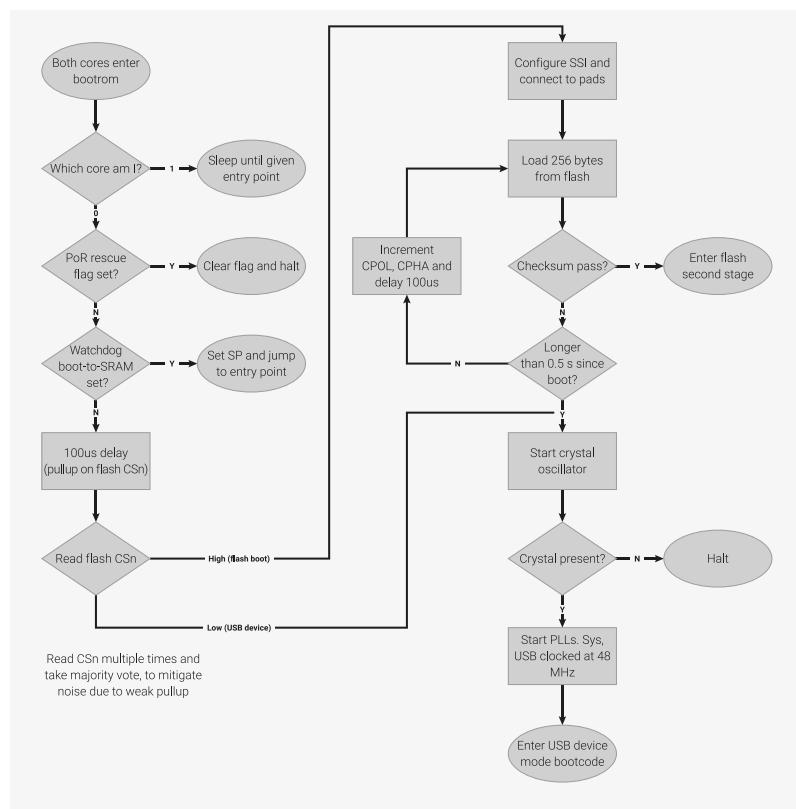
RP2040 Bootrom的完整源代码可于<https://github.com/raspberrypi/pico-bootrom>获取。

其中包含bootrom的第1版、第2版和第3版，分别对应B0、B1和B2硅片
修订版本。

2.8.1. 处理器控制的启动顺序

启动序列的流程图参见图15。

图15. RP2040
启动序列



在第2.7节所述的硬件控制启动序列之后，处理器控制启动序列开始执行：

- 两个处理器复位释放：均从相同地址进入ROM
- 处理器检查 SIO.CPUID
 - 处理器1进入休眠状态（启用SCR.SLEEPDEEP的WFE），并持续休眠，直至通过邮箱由用户代码唤醒
 - 处理器0继续从ROM执行
- 若上电事件源自救援DP，清除此标志并立即停止
 - 启发救援的调试主机将提供后续指令。
- 若看门狗擦写寄存器指示SRAM内存在预加载代码，则跳转至该代码

- 检查SPI CS引脚是否被拉低（“bootrom按钮”），若是则跳过闪存启动。
- 设置QSPI引脚的IO复用及管脚控制，初始化Synopsys SSI以使用标准SPI模式
- 发出XIP退出序列，以防闪存仍处于XIP模式且未断电
- 从 SPI 复制 256 字节至内部 SRAM (SRAM5)，并验证 CRC32 校验和的有效性
- 若校验和通过，则视所加载内容为有效的闪存第二阶段程序
- 开始从 SRAM (SRAM5) 执行已加载代码
- 若在尝试启动后 0.5 秒内未在 SPI 中找到有效映像，则切换至 USB 设备启动
- USB 设备启动：作为 USB 大容量存储设备出现
 - 可通过拖放 UF2 格式映像文件编程 SPI 闪存，或直接加载至 SRAM 并运行
 - 亦支持扩展的 PICOBLOCK 接口

2.8.1.1. 看门狗启动

看门狗启动允许用户安装自定义启动处理程序，并在非上电复位 (POR) 或欠压复位 (BOR) 时，将控制权从主启动序列分离同时简化了通过 JTAG 测试接口运行代码的操作系统识别写入看门狗高位 scratch 寄存器的以下数值：

- Scratch 4：魔术数字 `0xb007c0d3`
- Scratch 5：入口点与魔数异或 `-0xb007c0d3 (0x4ff83f2d)`
- Scratch 6：栈指针
- Scratch 7：入口点

如果任一魔数不匹配，则不会执行看门狗启动。若数字匹配，Bootrom 会在转移控制权前清零 scratch 4，以防该行为延续至后续重启。

2.8.1.2. 闪存启动序列

热启动闪存的主要挑战之一是强制外部闪存从 XIP 模式切换至可接受标准 SPI 命令的模式。针对未知闪存，无标准方法可中断 XIP。Bootrom 提供了兼容性广泛的尽力而为序列，具体如下：

- `CSn=1, IO[3:0]=4'b0000` （通过下拉电阻避免争用），发出 ×32 时钟
- `CSn=0, IO[3:0]=4'b1111` （通过上拉避免争用），发出 ×32 个时钟
- `CSn=1`
- `CSn=0, MOSI=1'b1` （低阻驱动，所有其他IO为高阻），发出 ×16 个时钟

此设计旨在避开Cypress、Micron及Winbond芯片上的XIP续传代码。如果设备已处于SPI模式，则将此序列解释为两个`FFh NOP`指令，应予以忽略。

鉴于此仅为尽力而为，可能存在某些设备固执地维持在XIP模式。因此有以下两种选择：

- 使用效率较低的XIP模式，其中每次传输均带有SPI指令前缀，使闪存设备保持SPI模式下的通信。
- 引导代码在SRAM中安装兼容的XIP退出序列，并配置看门狗，使得热启动时能直接跳转至该序列，放弃我们的预设序列。

发出XIP退出序列后，Bootrom尝试使用标准的 `03h`串行读取命令从闪存读取第二阶段代码，该命令几乎得到了普遍支持。由于Bootrom不可更改，其目标是实现兼容性

而非性能优化。

2.8.1.3. 闪存第二阶段

闪存第二阶段必须为SSI和外部闪存配置，以实现最佳的执行就地（execute-in-place）性能。这包括接口宽度、SCK频率、SPI指令前缀及用于地址-数据模式的XIP续传代码。通常可以对外部闪存执行特定操作，使其无需在每次访问时使用指令前缀，仅通过地址即可响应数据。

在SSI未正确配置对应闪存设备之前，无法通过XIP地址窗口访问闪存。此外，Synopsys SSI在未先禁用前无法进行重新配置。因此，第二阶段代码必须由Bootrom从闪存复制至SRAM，并在SRAM中执行。

或者，第二阶段可以简单地将外部闪存中的镜像映射（shadow）到SRAM中，而不配置执行原地（execute-in-place）。

这是第二阶段的唯一任务。所有其他芯片设置（如PLL、电压调节器）可由第二阶段执行完成后，通过XIP接口运行的平台初始化代码进行。

2.8.1.3.1. 校验和

从闪存加载的镜像的最后四个字节（预期为有效的闪存第二阶段）是前252个字节的CRC32校验和。校验和的参数如下：

- 多项式：`0x04c11db7`
- 输入反射：否
- 输出反射：否
- 初始值：`0xffffffff`
- 最终异或：`0x00000000`
- 校验和值以小端整数形式出现在镜像末尾。

Bootrom会进行约128次尝试，每次约4毫秒，总计约0.5秒，尝试失败后放弃，转而执行USB代码以不同SPI参数加载并校验第二阶段。如果检测到校验和通过，它将立即跳转到包含闪存第二阶段的252字节负载部分。

2.8.2. 在处理器核1上启动代码

如第2.8.1节介绍所述，复位后，处理器核心1将“休眠（启用SCR.SLEEPDEEP的WFE）”，并保持休眠状态，直到由用户代码通过邮箱唤醒。

如果您使用SDK，则可以直接调用`multicore_launch_core1`函数在处理器核心1上启动代码。
然而，本节描述了自行启动处理器核心1上代码的具体流程。

启动处理器核心1的过程涉及两个核心通过互处理器FIFO传递消息，以锁步方式驱动状态机运行。该状态机设计足够健壮，能够应对刚复位、并可能处于启动代码任意阶段（包括进入休眠状态）的处理器核心1。因此，该过程可在处理器核心1复位后任何时刻执行，无论是系统复位还是仅显式复位处理器核心1。

以下C代码为描述该过程的最简方式：

```
// 通过FIFO从核心0到核心1按顺序发送的值
//
// vector_table为VTOR寄存器的值
```

```

// sp为初始堆栈指针 (SP)
// entry为初始程序计数器 (PC) (请勿忘记设置Thumb位)
const uint32_t cmd_sequence[] =
{0, 0, 1, (uintptr_t)vector_table, (uintptr_t)sp, (uintptr_t)entry};

uint seq = 0;
do {
    uint cmd = cmd_sequence[seq];
    // 发送0之前应始终清空READ FIFO (来自核心1)
    if (!cmd) {
        // 丢弃READ FIFO中数据直至清空
        multicore_fifo_drain();
        // 执行SEV，核心1可能在等待FIFO空间
        __sev();
    }
    // 向写FIFO写入32位值
    multicore_fifo_push_blocking(cmd);
    // 从读FIFO读取32位值，一旦可用 uint32_t response = multicore_fifo_pop_blocking();
    // 在正确响应 (echo-d值) 时切换到下一个状态，否则重新开始
    seq = cmd == response ? seq + 1 : 0;
} while (seq < count_of(cmd_sequence));

```

2.8.3. 启动只读存储器内容

Bootrom的部分内容专门用于实现启动序列及USB启动接口。Bootrom中也包含对用户程序有用的代码。表163显示了Bootrom前几个固定存储单元的内存布局，这些内容对于定位Bootrom内的其他部分至关重要。

表163。Bootrom
内容在固定（众所
周知）地址处

地址	内容	描述
0x00000000	32位指针	初始启动堆栈指针
0x00000004	32位指针	指向启动复位处理函数的指针
0x00000008	32位指针	指向启动NMI处理函数的指针
0x0000000c	32位指针	指向启动硬故障处理函数的指针
0x00000010	'M', 'u', 0x01	魔术
0x00000013	字节	Bootrom 版本
0x00000014	16 位指针	指向公共函数查找表 (<code>rom_func_table</code>) 的指针
0x00000016	16 位指针	指向公共数据查找表 (<code>rom_data_table</code>) 的指针
0x00000018	16 位指针	指向辅助函数 (<code>rom_table_lookup()</code>) 的指针

2.8.3.1. Bootrom 函数

Bootrom 包含若干公共函数，这些函数提供了在设备缺乏其他代码时可能需要的有用 RP2040 功能，以及某些关键功能的高度优化版本，否则这些功能通常会占用大多数用户二进制文件的空间。

这些函数通常由 SDK 向用户提供，然而也提供了一种较低级的方法以定位它们（位置可能随每个 Bootrom 版本变更）并直接调用。

假设地址 `0x00000010`起始的三个字节为 ('M', 'u', 0x01)，则偏移量 `0x00000014`处的三个半字 (halfwords) 有效。

这三个值可用于动态定位 Bootrom 中的其他函数或数据。偏移量 `0x00000013` 处的版本字节仅供参考，不应作为推断任何函数精确位置的依据。

以下 SDK 代码示例展示了如何使用这三个 16 位指针查找其他函数或数据。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_bootrom/bootrom.c 第 12 至 19 行

```

12 void *rom_func_lookup(uint32_t code) {
13     return rom_func_lookup_inline(code);
14 }
15
16 void *rom_data_lookup(uint32_t code) {
17     return rom_data_lookup_inline(code);
18 }
```

参数 `code` 对应下表中的 `CODE` 值，计算方式如下：

```

uint32_t rom_table_code(char c1, char c2) {
    return (c2 << 8) | c1;
}
```

2.8.3.1.1. 快速位计数／操作函数

这些是常用位计数／操作函数的优化版本。

通常情况下，您无需直接调用这些方法，因为 SDK 中的 `pico_bit_ops` 库默认替换了相应的标准编译器库函数，标准函数如 `_builtin_popcount` 或 `_clzdi2` 会自动调用对应的 Bootrom 实现（详情请参见 `pico_bit_ops`）。

这些函数在 Bootrom 版本 1 (V1) 与版本 2 (V2) 之间的执行速度略有差异。

表164。快速位计数／操作函数。

代码	平均周期数 V1	平均周期数 V2/V3	描述
'P','3'	18	20	<code>uint32_t _popcount32(uint32_t value)</code>
			返回 <code>value</code> 中值为 1 的位数。
'R','3'	21	22	<code>uint32_t _reverse32(uint32_t value)</code>
			返回 <code>value</code> 的位逆序。
'L','3'	13	9.6	<code>uint32_t _clz32(uint32_t value)</code>
			返回 <code>value</code> 中连续的高位 0 的数量。如果 <code>value</code> 为零，则返回 32。
'T','3'	12	11	<code>uint32_t _ctz32(uint32_t value)</code>
			返回 <code>value</code> 中连续的低位 0 的数量。如果 <code>value</code> 为零，则返回 32。

2.8.3.1.2 快速批量内存填充／复制函数

这些函数是大多数语言运行时通常提供的高度优化的批量内存填充和复制函数。

通常情况下，您无需直接调用这些方法，因为 SDK 中的 `pico_mem_ops` 库默认替换了相应的标准 ARM EABI 函数，使得标准 C 库函数如 `memcpy` 或 `memset` 自动使用 Bootrom 实现（详见 `pico_mem_ops`）。

表 165。优化的批量内存填充 /
复制函数

代码	描述
'M', 'S'	<code>uint8_t * _memset(uint8_t *ptr, uint8_t c, uint32_t n)</code> 将从 <code>ptr</code> 开始的 <code>n</code> 个字节设置为值 <code>c</code> ，返回 <code>ptr</code> 。
'S', '4'	<code>uint32_t * _memset4(uint32_t *ptr, uint8_t c, uint32_t n)</code> 将从 <code>ptr</code> 开始的 <code>n</code> 个字节设置为值 <code>c</code> ，返回 <code>ptr</code> 。注意，此函数为 <code>_memset</code> 的稍高效变体，仅当 <code>ptr</code> 按字对齐时方可使用。
'M', 'C'	<code>uint8_t * _memcpy(uint8_t *dest, uint8_t *src, uint32_t n)</code> 将从 <code>src</code> 开始的 <code>n</code> 个字节复制到 <code>dest</code> ，返回 <code>dest</code> 。若内存区域重叠，结果未定义。
'C', '4'	<code>uint8_t * _memcpy44(uint32_t *dest, uint32_t *src, uint32_t n)</code> 将从 <code>src</code> 开始的 <code>n</code> 个字节复制到 <code>dest</code> ，返回 <code>dest</code> 。若内存区域重叠，结果未定义。 注意，此函数为 <code>_memcpy</code> 的稍高效变体，仅当 <code>dest</code> 和 <code>src</code> 均按字对齐时方可使用。

2.8.3.1.3. 闪存访问函数

这些是底层的闪存辅助函数。

表 166. 闪存
访问函数

代码	描述
'I', 'F'	<code>void _connect_internal_flash(void)</code> 将所有 QSPI 引脚控制恢复至默认状态，并将 SSI 连接至 QSPI 引脚。
'E', 'X'	<code>void _flash_exit_xip(void)</code> 首先将 SSI 设置为串行模式操作，然后执行第 2.8.1.2 节中描述的固定 XIP 退出序列。请注意，bootrom 代码采用 IO 强制逻辑驱动 CS 引脚，返回 SSI 至 XIP 模式前必须清除此状态（例如通过调用 <code>_flash_flush_cache</code> 函数）。此函数将 SSI 配置为固定的 SCK 时钟分频器，分频值为 /6。
'R', 'E'	<code>void _flash_range_erase(uint32_t addr, size_t count, uint32_t block_size, uint8_t block_cmd)</code> 从 <code>addr</code> （相对于闪存起始的偏移）开始，擦除 <code>count</code> 字节。可选地，传入块擦除命令，例如 D8h 块擦除，以及该命令擦除的块大小——本函数将在可能的情况下使用更大块擦除，以极大提升擦除速度。 <code>addr</code> 必须按 4096 字节扇区对齐，且 <code>count</code> 必须为 4096 字节的整数倍。
'R', 'P'	<code>void flash_range_program(uint32_t addr, const uint8_t *data, size_t count)</code> 将 <code>data</code> 编程至以 <code>addr</code> （相对于闪存起始的偏移）为起始地址、大小为 <code>count</code> 字节的闪存区域。 <code>addr</code> 必须按 256 字节边界对齐，且 <code>count</code> 须为 256 的整数倍。
'F', 'C'	<code>void _flash_flush_cache(void)</code> 刷新并启用 XIP 缓存。同时清除 QSPI CSn 上的 IO 强制，使 SSI 可正常驱动闪存芯片选择信号。
'C', 'X'	<code>void _flash_enter_cmd_xip(void)</code> 配置 SSI 以生成标准的 03h 串行读取命令，包含 24 位地址，于每次 XIP 访问时触发。此为速度极慢的 XIP 配置，但具有广泛的兼容性。调试器在执行闪存擦除/编程操作后调用此函数，以使新编程的代码和数据对调试主机可见，无需准确了解所连接的闪存设备类型。

从用户代码擦除闪存扇区的典型调用顺序如下：

- `_connect_internal_flash`
- `_flash_exit_xip`
- `_flash_range_erase(addr, 1 << 12, 1 << 16, 0xd8)`
- `_flash_flush_cache`
- 调用`_flash_enter_cmd_xip`, 或调用先前复制到SRAM的闪存二级启动程序。

请注意，在该序列的首次与末次调用间，SSI处于无法处理XIP访问的状态，因此调用中间函数的代码必须驻留于SRAM中。SDK中的[hardware_flash](#)库封装了相关细节。

2.8.3.1.4. 调试支持功能

这两种方法简化了在设备上调用代码并将控制权返回调试器的流程。

表167。调试支持功能

代码	描述
'D', 'T'	<p><code>_debug_trampoline</code></p> <p>用于返回时设置断点的简易调试跳板。</p> <p>此方法协助调试器在不设置硬件断点的情况下调用ROM例程。函数地址通过 <code>r7</code> 传递，参数通过 <code>r0</code> 至 <code>r3</code> 按 ABI 规定传递。</p> <p>此方法不返回，结束时执行 <code>BKPT #0</code> 指令。</p>
'D', 'E'	<p><code>_debug_trampoline_end</code></p> <p>此地址为 <code>debug_trampoline</code> 中最后一条 <code>BKPT #0</code> 指令的位置。可通过与程序计数器比较，检测 <code>debug_trampoline</code> 调用的完成。</p>

2.8.3.1.5. 杂项功能

这些剩余功能未归类于其他类别，且通过[pico_bootrom](#)库在SDK中提供（详见[pico_bootrom](#)）。

表168。
杂项
函数

代码	描述
'U', 'B'	<p><code>void _reset_to_usb_boot(uint32_t gpio_activity_pin_mask, uint32_t disable_interface_mask)</code></p> <p>重置RP2040并利用看门狗功能重新启动至BOOTSEL模式：</p> <ul style="list-style-type: none"> • <code>gpio_activity_pin_mask</code> 用于通过连接至GPIO的LED启用USB大容量存储设备的“活动指示灯”： <ul style="list-style-type: none"> ◦ 0 不使用任何引脚，等同冷启动。 ◦ 否则为单个位掩码，指示应将哪个GPIO引脚设置为输出，并在主机进行大容量存储活动时将其置高。 • <code>disable_interface_mask</code> 用于控制暴露的USB接口： <ul style="list-style-type: none"> ◦ 0 启用两个接口（等同冷启动） ◦ 1 禁用USB大容量存储接口（参见第2.8.4节） ◦ 2 禁用USB PICOBOT 接口（参见第2.8.5节）

'W', 'V'	<code>_wait_for_vector</code> 此方法由核心1在复位时进入，等待核心0发起启动。极少数情况下应调用此方法（重置核心1更为合适）。此方法不会返回，且应仅在核心1上调用。
'E', 'C'	已废弃 请勿使用此函数，因其可能不存在。

2.8.3.2 快速浮点库

Bootrom 包含经过优化的单精度浮点实现。此外，从 V2 版本开始，还包含经过优化的双精度浮点实现。每种精度对应的函数指针保存在通过 `rom_data_lookup` 表查找的表结构中（详见第 2.8.3.3 节）。

2.8.3.2.1 实现细节

速度与体积之间始终存在权衡。尽管浮点例程的总体目标是在较小体积内实现优良性能，重点更侧重于基本运算（加、减、乘、除及平方根）的性能提升，以及科学函数（三角函数、对数及指数函数）的体积压缩。

整个过程中采用 IEEE 单精度和双精度数据格式，但为了减小代码体积，输入的非规格化数被视为零，输入的 NaN 被视为无穷大，输出的非规格化数被冲洗为零，输出的 NaN 被渲染为无穷大。仅支持四舍五入至最近且遇5取偶的舍入模式。不支持触发异常。

五种基本运算返回的结果始终正确舍入。

科学函数的返回结果始终在精确值的 1 ULP（单位最后位）误差范围内。在许多情况下，结果更为准确。

科学函数使用内部定点表示进行计算，因此在将结果转换回浮点数时，如需大量规整移位的情况下，准确度（以 ULP 误差衡量，而非绝对误差）较差。例如，计算接近 π 倍数的正弦值、接近 $\pi/2$ 奇数倍的余弦值或接近 1 的对数值时，会出现此情形。当结果极大时，正切函数的精度亦较差。虽然覆盖这些情况是可行的，但会显著增加代码体积，且极少有程序类型在此类情形下对精度有严格要求。

正弦、余弦及正切函数仅在有限范围内正确运行：单精度参数 x 的范围为 $-128 < x < +128$ ，双精度参数 x 的范围为 $-1024 < x < +1024$ 。此设计旨在避免代码中（至少实际效果上）存储高精度 π 值，因而节省代码空间。若需进行更广泛参数范围的准确约简，可在库外完成，但此类需求极为罕见。

① 注意

SDK 的 cos/sin 函数执行该范围约简，因此可接受全范围参数，但对于超出该范围的输入，运算速度较慢。

2.8.3.2.2. 函数

这些函数遵循标准 ARM EABI 的浮点数传递规范。

您无需直接调用这些方法，因为默认使用的 SDK `pico_float` 和 `pico_double` 库已替代 ARM EABI `Float` 函数，使得 C/C++ 级别代码（或间接受 C 语言实现的语言，如 `MicroPython`）自动调用这些 Bootrom 函数以执行相应的浮点运算。

某些函数的行为与对应的 C 标准库函数并不完全相同。因此，强烈建议在使用 SDK 时，直接使用常规的 `math.h` 函数或 `pico/float.h` 及 `pico/double.h` 中的函数，而非尝试直接调用 Bootrom。

请注意，Bootrom 第一版（V1）不支持双精度浮点，但 SDK 中上述的 `pico_double` 库会自动引入 V1 所需的额外代码。

注意

有关在 SDK 中使用浮点数及其实际运行时间的详细信息（同时注意部分转换函数已被 SDK 重新实现以提升速度），请参阅浮点支持章节。

表169。单精度浮点函数表。

时序为随机（最差情况）输入
入下的平均时间，
单位为微秒。时序
显示为N/A的函数
在该ROM版本中
不存在，函数
指针应视为无
效。偏移量0x54及
之后的函数（及表
项）仅存在于V2
ROM版本中。

偏移量	V1周期 (平均)	V2/V3 周期 (平均)	描述
所有引导ROM版本通用函数			
0x00	71	71	float _fadd(float a, float b) 返回 $a + b$
0x04	74	74	float _fsub(float a, float b) 返回 $a - b$
0x08	69	58	float _fmul(float a, float b) 返回 $a * b$
0x0c	71	71	float _fdiv(float a, float b) 返回 a / b
0x10	不适用	不适用	已废弃 请勿使用此函数
0x14	不适用	不适用	已废弃 请勿使用此函数
0x18	63	63	float _fsqrt(float v) 如果 v 为负数，则返回正无穷或负无穷。（注意 V1 在此情况下返回正无穷）
0x1c	37	40	int _float2int(float v) 将浮点数转换为有符号整数，向负无穷方向舍入，并将结果限制在范围 <code>-0x80000000</code> 至 <code>0x7FFFFFFF</code>
0x20	36	39	int _float2fix(float v, int n) 将浮点数转换为有符号定点整数表示，其中 n 指定二进制小数点在结果定点表示中的位置，例如 <code>_float2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 <code>-0x80000000</code> 至 <code>0x7FFFFFFF</code>
0x24	38	39	uint _float2uint(float v) 将浮点数转换为无符号整数，向负无穷方向舍入，并将结果限制在范围 <code>0x00000000</code> 至 <code>0xFFFFFFFF</code>

0x28	38	38	<code>uint _float2ufix(float v, int n)</code> 将浮点数转换为无符号定点整数表示，其中 n 指定二进制小数点在结果定点表示中的位置，例如 <code>_float2ufix(0.5f, 16) == 0x8000</code> 。该方法向 $-\infty$ 方向舍入，且将结果整数限制在范围 0x00000000 至 0xFFFFFFFF
0x2c	55	55	<code>float _int2float(int v)</code> 将有符号整数转换为最接近的浮点值，平分情况采用偶数舍入
0x30	53	53	<code>float _fix2float(int32_t v, int n)</code> 将有符号定点整数表示转换为最接近的浮点值， 平分情况采用偶数舍入。 n 指定定点数中二进制小数点的位置， 因此 $= \text{nearest}(v / 2^n)$
0x34	54	54	<code>float _uint2float(uint32_t v)</code> 将无符号整数转换为最接近的浮点值，平分情况采用偶数舍入
0x38	52	52	<code>float _ufix2float(uint32_t v, int n)</code> 将无符号定点整数表示转换为最接近的浮点数值， 平分情况采用偶数舍入。 n 指定定点数中二进制小数点的位置， 因此 $= \text{nearest}(v / 2^n)$
0x3c	603	587	<code>float _fcos(float angle)</code> 返回 $angle$ 的余弦值。 $angle$ 的单位为弧度，且必须处于 -128 至 128 范围内
0x40	593	577	<code>float _fsin(float angle)</code> 返回 $angle$ 的正弦值。 $angle$ 的单位为弧度，且必须处于 -128 至 128 范围内
0x44	669	653	<code>float _ftan(float angle)</code> 返回 $angle$ 的正切值。 $angle$ 的单位为弧度，且必须处于 -128 至 128 范围内
0x48	不适用	不适用	已废弃 请勿使用此函数
0x4c	542	524	<code>float _fexp(float v)</code> 返回 v 的指数值，即 e 的 v 次幂 e^v
0x50	810	789	<code>float _fln(float v)</code> 返回 v 的自然对数。如果 $v <= 0$ 返回 -无穷大
仅存在于 V2/V3 启动只读存储器中的函数（及表项）			
0x54	不适用	25	<code>int _fcmp(float a, float b)</code> 比较两个浮点数，返回： <ul style="list-style-type: none">• 0，如果 $a == b$• -1，如果 $a < b$• 1，如果 $a > b$
0x58	不适用	667	<code>float _fatan2(float y, float x)</code> 计算 y/x 的反正切，通过参数符号确定正确象限

0x5c	不适用	62	<code>float _int642float(int64_t v)</code> 将有符号64位整数转换为最接近的浮点数值，平分时采用偶数舍入法
0x60	不适用	60	<code>float _fix642float(int64_t v, int n)</code> 将有符号定点64位整数表示转换为最接近的浮点数值，平分时采用偶数舍入法。n指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x64	不适用	58	<code>float _uint642float(uint64_t v)</code> 将无符号64位整数转换为最接近的浮点值，平局时四舍六入
0x68	不适用	57	<code>float _ufix642float(uint64_t v, int n)</code> 将无符号定点64位整数表示转换为最接近的浮点值，平局时四舍六入。n指定定点数中二进制点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x6c	不适用	54	<code>_float2int64</code> 将浮点数转换为有符号64位整数，采用向负无穷舍入，并将结果限制在范围-0x8000000000000000至0x7FFFFFFFFFFFFF
0x70	不适用	53	<code>_float2fix64</code> 将浮点数转换为有符号定点64位整数表示，其中n指定结果定点表示的二进制点位置 - 例如： <code>_float2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷舍入，且将结果整数限定在范围-0x8000000000000000至0x7FFFFFFFFFFFFF
0x74	不适用	42	<code>_float2uint64</code> 将浮点数转换为无符号64位整数，向-∞舍入，并将结果限制在范围0x0000000000000000至0xFFFFFFFFFFFFFF
0x78	不适用	41	<code>_float2ufix64</code> 将浮点数转换为无符号定点64位整数表示，其中n指定结果定点表示中二进制点的位置， 例如 <code>_float2ufix(0.5f, 16) == 0x8000</code> 。此方法向-∞舍入，并将结果整数限制在范围0x0000000000000000至0xFFFFFFFFFFFFFF
0x7c	不适用	15	<code>double _float2double(float v)</code> 将浮点数转换为双精度浮点数
此函数仅存在于V3启动只读存储器中			
0x48 (使用先前已弃用的槽位)	<small>577 (仅适用于V3)</small>		<code>float (float) _fsincos(float 角度)</code>
			计算指定角度的正弦和余弦。角度以弧度表示，且必须位于-128至128范围内。 正弦值存储于寄存器r0（因此为官方函数返回值），余弦值存储于寄存器r1。该方法比分别调用 <code>_fsin</code> 和 <code>_fcos</code> 显著更快。

请注意，V2/V3启动ROM包含等效的双精度浮点运算函数表。

偏移量相同，但原为float的位置改为double（反之亦然，用于float<>double转换）。

表 170。双精度浮点函数表。

时序为随机（最差情况）输入下的平均时间，单位为微秒。时序显示为N/A的函数在该ROM版本中不存在，函数指针应视为无效。偏移量自0x54起的函数及表项仅存在于V2和V3 ROM中。

偏移量	周期数 (平均值) *	描述
0x00	91	<code>double _dadd(double a, double b)</code> 返回 $a + b$
0x04	95	<code>double _dsub(double a, double b)</code> 返回 $a - b$
0x08	155	<code>double _dmul(double a, double b)</code> 返回 $a * b$
0x0c	183	<code>double _ddiv(double a, double b)</code> 返回 a / b
0x10	不适用	已废弃 请勿使用此函数
0x14	不适用	已废弃 请勿使用此函数
0x18	169	<code>double _dsqrt(double v)</code> 如果 v 为负，则返回 0 或 $-\infty$ 。
0x1c	75	<code>int _double2int(double v)</code> 将 double 转换为有符号整数，向 $-\infty$ 舍入，并将结果限制在范围 $0x8000000$ 至 $0x7FFFFFFF$
0x20	74	<code>int _double2fix(double v, int n)</code> 将 double 转换为有符号定点整数表示，其中 n 指定结果中二进制小数点的位置——例如 <code>_double2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 $0x8000000$ 至 $0x7FFFFFFF$
0x24	63	<code>uint _double2uint(double v)</code> 将 double 转换为无符号整数，向 $-\infty$ 舍入，并将结果限制在范围 $0x0$ 至 $0xFFFFFFFF$
0x28	62	<code>uint _double2ufix(double v, int n)</code> 将 double 转换为无符号定点整数表示，其中 n 指定结果中二进制小数点的位置，例如 <code>_double2ufix(0.5f, 16) == 0x8000</code> 。该方法向 $-\infty$ 方向舍入，且将结果整数限制在范围 $0x00000000$ 至 $0xFFFFFFFF$
0x2c	69	<code>double _int2double(int v)</code> 将有符号整数转换为最接近的 double 值，平分时向偶数舍入
0x30	68	<code>double _fix2double(int32_t v, int n)</code> 将有符号定点整数表示转换为最接近的 double 值，平分时向偶数舍入。 n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x34	64	<code>double _uint2double(uint32_t v)</code> 将无符号整数转换为最接近的 double 值，平分时向偶数舍入

偏移量	周期数 (平均值) *	描述
0x38	62	<p><code>double _ufix2double(uint32_t v, int n)</code></p> <p>将无符号定点整数表示转换为最接近的 double 值，平分时向偶数舍入。n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x3c	1617	<p><code>double _dcos(double angle)</code></p> <p>返回 angle 的余弦值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x40	1618	<p><code>double _dsin(double angle)</code></p> <p>返回 angle 的正弦值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x44	1891	<p><code>double _dtan(double angle)</code></p> <p>返回 angle 的正切值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x48	不适用	<p>已废弃</p> <p>请勿使用此函数</p>
0x4c	804	<p><code>double _dexp(double v)</code></p> <p>返回 v 的指数值，即 e 的 v 次幂 e^v</p>
0x50	428	<p><code>double _dln(double v)</code></p> <p>返回 v 的自然对数。如果 $v \leq 0$ 返回 -无穷大</p>
0x54	39	<p><code>int _dcmp(double a, double b)</code></p> <p>比较两个浮点数，返回：</p> <ul style="list-style-type: none"> • 0，如果 $a == b$ • -1，如果 $a < b$ • 1，如果 $a > b$
0x58	2168	<p><code>double _datan2(double y, double x)</code></p> <p>计算 y/x 的反正切，通过参数符号确定正确象限</p>
0x5c	55	<p><code>double _int642double(int64_t v)</code></p> <p>将有符号64位整数转换为最接近的double值，遇平局时向偶数舍入</p>
0x60	56	<p><code>double _dix642double(int64_t v, int n)</code></p> <p>将有符号64位整数表示转换为最接近的double值，遇平局时向偶数舍入。n指定定点表示中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x64	50	<p><code>double _uint642double(uint64_t v)</code></p> <p>将无符号64位整数转换为最接近的double值，遇平局时向偶数舍入</p>
0x68	49	<p><code>double _ufix642double(uint64_t v, int n)</code></p> <p>将无符号定点64位整数表示转换为最接近的double值，平分时向偶数舍入。n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x6c	64	<p><code>_double2int64</code></p> <p>将double转换为有符号64位整数，向负无穷大舍入，并将结果限定在范围-0x8内0000000000000000至0x7FFFFFFFFFFFFF</p>

偏移量	周期数 (平均值) *	描述
0x70	63	<code>_double2fix64</code> 将 double 转换为有符号定点 64 位整数表示， <i>n</i> 指定结果定点表示中二进制小数点的位置——例如： <code>_double2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 0x8000000000000000 至 0x7FFFFFFFFFFFFF
0x74	53	<code>_double2uint64</code> 将 double 转换为无符号 64 位整数，向 -∞ 舍入，并将结果限制在 0x 范围内 0000000000000000 至 0xFFFFFFFFFFFFFF
0x78	52	<code>_double2ufix64</code> 将 double 转换为无符号定点 64 位整数表示，其中 <i>n</i> 指定二进制点在最终定点表示中的位置，例如 <code>_double2ufix(0.5f, 16) == 0x8000</code> 。该方法向 -∞ 方向舍入，且将结果整数限制在范围 0x0000000000000000 至 0xFFFFFFFFFFFFFF
0x7c	23	<code>float _double2float(double v)</code> 将 double 转换为 float
此函数仅存在于 V3 启动只读存储器中		
0x48 (使用先前已弃用的槽位)	1718 (仅限 V3)	<code>double (,double) _sincos(double angle)</code> 计算指定角度的正弦和余弦。 <i>angle</i> 单位为弧度，且必须位于 -1024 至 1024 范围内。正弦值返回至寄存器 <i>r0/r1</i> （因此为官方返回值），余弦值返回至寄存器 <i>r2/r3</i> 。此方法明显快于分别调用 <code>_sin</code> 和 <code>_cos</code> 。

2.8.3.3. Bootrom 数据

Bootrom 数据表 (`rom_data_table`) 包含以下指针。

表 171. Bootrom 数据指针

代码	值 (16 位指针) 描述
'C', 'R'	<code>const char *copyright_string</code> Raspberry Pi Trading Ltd 的版权声明字符串。
'G', 'R'	<code>const uint32_t *git_revision</code> Bootrom git 修订版的最高 8 位十六进制数字。
'F', 'S'	<code>fplib_start</code> 浮点库代码和数据的起始地址。该地址与 <code>fplib_end</code> 及 <code>soft_float_table</code> 中的各函数指针一起，可用于将浮点实现复制至 RAM（如需）。
'S', 'F'	<code>soft_float_table</code> 有关本表内容，详见表 169。
'F', 'E'	<code>fplib_end</code> 浮点库代码和数据的结束地址。

'S','D'	<code>soft_double_table</code>
此条目仅存在于 V2 版本 bootrom 中。有关本表内容, 请参见表170。	
'P','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'R','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'L','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'T','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。

2.8.4. USB大容量存储设备接口

引导ROM提供了标准USB引导加载程序, 使得可通过 *UF2*文件 (见第2.8.4.2节) 将代码复制到RP2040的可写驱动器。

复制到驱动器的 *UF2*文件将被下载并写入闪存或RAM, 设备自动重启, 从而仅凭USB连接即可轻松下载并运行RP2040上的代码。

2.8.4.1. RPI-RP2驱动器

RP2040以名为 *RPI-RP2*的标准128MB闪存驱动器形式出现, 格式为单一FAT16分区。驱动器上始终仅显示两个实际文件。

- `INFO_UF2.TXT` — 包含UF2引导加载程序及版本的字符串描述。
- `INDEX.HTM` - 重定向至关于RP2040设备的信息。

主机可以向USB驱动器写入任何类型的文件, 但通常这些文件不会被实际存储, 仅因主机端缓存而呈现为已存储状态。

当向设备写入 *UF2*文件时, 设备会识别其特殊内容, 并将数据写入RAM或Flash的指定位置。在整个有效 *UF2*文件下载完成后, RP2040会自动重启以运行新下载的代码。

ⓘ 注意

当前 `INDEX.HTM`文件重定向至 <https://www.raspberrypi.com/documentation/microcontrollers/>

2.8.4.2. UF2格式详情

💡 提示

要生成 *UF2*文件, 请使用picotool中的 *UF2*转换功能。

ⓘ 注意

无效的 *UF2*文件可能无法写入, 或仅部分写入RP2040后即失败。并非所有操作系统都会在写入失败后通知磁盘写入错误。您可以使用 `picotool`验证 *UF2*文件是否已正确写入RP2040。

- 发送至设备的所有数据必须包含在带有familyID字段且该字段值设为0xe48bff56的UF2数据块内, 且payload_size须为256。
- 所有数据必须定位在 (且完全位于) 以下内存范围内, 具体取决于第一个遇到的 *UF2*数据块地址, 该地址决定所下载二进制文件的类型:

a. 常规闪存二进制文件

- **0x10000000-0x11000000**闪存：所有数据块必须按照256字节对齐。写入超出物理闪存末端的部分将回绕至闪存起始位置。

b. 仅限RAM的二进制文件

- **0x20000000-0x20042000**主RAM：数据块可按字节对齐任意定位。
- **0x15000000-0x15004000**闪存缓存：鉴于未定位闪存，闪存缓存可用作具备与主RAM相同属性的RAM。

i 注意

传统上，UF2仅用于写入Flash，但这主要是由于采用无元数据的.BIN文件作为生成UF2文件源的限制。RP2040充分利用了UF2的固有灵活性，支持构建生成的更丰富的.ELF格式的完整二进制文件系列，作为生成UF2文件的源文件。

- numBlocks必须指定一个符合上述区域大小限制的二进制文件总大小。
- 更改 numBlocks或二进制类型（由UF2块的目标地址决定）将放弃当前正在进行的传输。
- 所有数据必须包含在未标记UF2_FLAG_NOT_MAIN_FLASH的区块中，该标记指示应忽略的内容，而非Flash与RAM的区分。

Flash始终以4kB扇区为单位擦除，因此，在包含Flash二进制UF2中某一扇区内部分256字节页的数据时，该扇区剩余的256字节页面将被擦除，但其内容未定义。RP2040启动只读存储器将接受包含此类部分填充扇区的UF2二进制文件，但因一个缺陷（RP2040-E14），若存在除末尾以外的任何部分填充扇区，可能导致此类二进制文件无法正确写入。大多数闪存二进制文件均为4kB对齐且连续，因此通常只有最后一个扇区是部分填充的。若需向闪存写入非对齐或非连续的UF2文件，应确保除最后一个扇区外，所有将写入的扇区均包含完整的4kB数据。从SDK版本1.3.1起，elf2uf2工具会自动处理此问题，明确向适当的部分填充扇区添加填充零页。

当在一次有效传输过程中，numBlocks个区块中的每一个至少被接收一次时，该二进制文件即被视为“已下载”。块的数据仅在主机重传重复块时首次写入。

下载常规闪存二进制文件后，将执行复位，随后通过bootrom进入闪存二进制第二阶段（地址0x1000000-闪存起始地址）（如有效）。

下载的仅RAM二进制文件通过看门狗复位进入二进制起始地址，该地址被计算为下载块中最低的地址（当主RAM和闪存缓存共存时，主RAM地址视为较低地址）。

主机软件可以通过PICOBLOCK接口临时禁用UF2写入操作，以防止对该接口正在执行的操作产生干扰（详见下文），在此过程中，任何进行中的UF2文件写入将被中止。

2.8.5. USB PICOBLOCK接口

PICOBLOCK接口是一种在RP2040处于BOOTSEL模式时用于交互的低级USB协议。该接口可与USB大容量存储接口同时使用。

该功能支持灵活地对RAM或Flash进行读写、设备重启、代码执行及多项其他管理操作。

接口相关的常量及结构定义载于SDK头文件：https://github.com/raspberrypi/pico-sdk/blob/master/src/common/boot_picoblock_headers/include/boot/picoblock.h

2.8.5.1. 设备识别

RP2040设备通过其设备描述符中的*Vendor ID*和*Product ID*进行识别（详见表172）。

表172。RP2040
启动设备
描述符

字段	数值
bLength	18
bDescriptorType	1
bcdUSB	1.10
bDeviceClass	0
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
idVendor	0x2e8a
idProduct	0x0003
bcdDevice	1.00
iManufacturer	1
iProduct	2
iSerial	3
bNumConfigurations	1

2.8.5.2 识别接口

PICOBOOT接口由「厂商专用（Vendor Specific）」接口类、零接口子类及接口协议识别（见表173）。请注意，不应依赖接口编号，该编号取决于设备是否同时暴露大容量存储接口。还应注意，设备可能根本未暴露PICOBOOT接口，因此不得假设其存在。

表173 PICOBOOT
接口描述符

字段	数值
bLength	9
bDescriptorType	4
bInterfaceNumber	可变
bAlternateSetting	0
bNumEndpoints	2
bInterfaceClass	0xff（厂商专用）
bInterfaceSubClass	0
bInterfaceProtocol	0
ilInterface	0

2.8.5.3. 识别端点

PICOBOOT接口提供一个单独的*BULK OUT*端点和一个单独的*BULK IN*端点。这些端点可根据其方向和类型进行识别。不应依赖端点编号。

2.8.5.4. PICOBEST命令

两个bulk端点用于发送命令并检索成功的命令结果。所有命令均为32字节（参见表174），并通过BULK OUT端点发送。

表174。PICOBEST命令定义

偏移量	名称	描述
0x00	dMagic	值 0x431fd10b
0x04	dToken	用户提供的令牌，用于标识该请求
0x08	bCmdId	命令ID。注意，最高位表示数据传输方向（0x80 = IN）
0x09	bCmdSize	args字段中有效数据的字节数
0x0a	保留	0x0000
0x0c	dTransferLength	主机预期通过批量通道发送或接收的字节数
0x10	args	16字节特定命令数据，以零填充

若发送的命令无效或无法识别，批量端点将被阻塞。更多信息可通过GET_COMMAND_STATUS请求获得（见第2.8.5.5.2节）。

在初始32字节数据包之后，若dTransferLength非零，则通过批量传输管道传输相应字节数的数据，命令随后以相反方向的空数据包完成。若dTransferLength为零，则通过空的IN数据包表示命令成功。

支持以下命令（为简洁起见，省略公共字段dMagic、dToken及reserved）

2.8.5.4.1. EXCLUSIVE_ACCESS (0x01)

声明或释放通过USB对RP2040进行写入的排他访问权限（区别于大容量存储接口）

表175。PICOBEST排他访问命令结构

偏移量	名称	数值 / 描述	
0x08	bCmdId	0x01 (EXCLUSIVE_ACCESS)	
0x09	bCmdSize	0x01	
0x0c	dTransferLength	0x00000000	
0x10	bExclusive	NOT_EXCLUSIVE (0)	对USB大容量存储操作无限制
		EXCLUSIVE (1)	禁用USB大容量存储写入（主机应将其视为写保护失败，但任何活动的UF2下载将被中止）
		EXCLUSIVE_AND_EJECT (2)	通过标记驱动器媒体为不存在（弹出驱动器）来锁定USB大容量存储接口

2.8.5.4.2. REBOOT (0x02)

使RP2040从BOOTSEL模式重启。注意，若重启至闪存且未找到有效的第二阶段引导程序，可能会重新进入BOOTSEL模式。

表176。PICOBEST重启访问命令结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x02（重启）
0x09	bCmdSize	0x0c

0x0c	dTransferLength	0x00000000	
0x10	dPC	开始执行的地址。有效值包括：	
		0x00000000 通过标准Flash启动机制进行重启	
		RAM地址 通过看门狗重启，并从RAM中指定地址开始执行	
0x14	dSP	重启后的初始堆栈指针（仅在引导进入RAM时使用）	
0x18	dDelayMS	重启前的延迟时间（毫秒）	

2.8.5.4.3. FLASH_ERASE (0x03)

擦除一段连续的Flash扇区。

表177. PICOBOOT
Flash擦除命令
结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x03 (FLASH_ERASE)
0x09	bCmdSize	0x08
0x0c	dTransferLength	0x00000000
0x10	dAddr	要擦除的Flash地址，从该地址起始。此值必须以扇区（4kB）为对齐单位。
0x14	dSize	要擦除的字节数。此值必须为扇区（4kB）的整数倍。

2.8.5.4.4. 读取 (0x84)

从RP2040读取连续的内存（Flash、RAM或ROM）范围。

表178. PICOBOOT
读取内存
命令（Flash、
RAM、ROM）结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x84 (读取)
0x09	bCmdSize	0x08
0x0c	dTransferLength	必须与dSize一致。
0x10	dAddr	读取的起始地址。地址可以位于Flash、RAM或ROM中。
0x14	dSize	要读取的字节数。

2.8.5.4.5. 写入 (0x05)

在RP2040上写入连续的内存范围（Flash或RAM）。

表179。PICOBOOT
写入内存
命令（Flash、
RAM）结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x05 (写入)
0x09	bCmdSize	0x08
0x0c	dTransferLength	必须与dSize一致。

偏移量	名称	数值 / 描述
0x10	dAddr	写入地址。写入地址可位于Flash或RAM中，但若位于Flash，则须按页（256字节）对齐。请注意，Flash必须先行擦除，否则结果无法确定。
0x14	dSize	写入的字节数。若写入Flash且大小非页（256字节）整数倍，则最后一页将以零填充至页尾。

2.8.5.4.6. EXIT_XIP (0x06)

退出Flash XIP模式。首先初始化SSI以支持串行传输，然后依据第2.8.1.2节，发出XIP退出序列，尝试使Flash响应标准串行SPI命令。SSI配置为固定的时钟分频器/6，因而USB引导程序将以8MHz频率驱动SCLK。

表180。PICOBOOT
退出执行就地
(XIP) 命令
结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x06 (EXIT_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.7. ENTER_XIP (0x07)

进入闪存XIP模式。该配置使SSI发出标准的03h串行读取命令，每次XIP访问包含24个地址时钟和32个数据时钟。这是一种速度较慢但被广泛支持的闪存读取方式。此功能旨在使闪存易于访问（即仅需访问0x10……段地址），无需了解连接的闪存具体类型。该模式适合从闪存执行代码，但速度远低于例如QSPI XIP访问。

表181。PICOBOOT
进入执行就地
(XIP)
命令

偏移量	名称	数值 / 描述
0x08	bCmdId	0x07 (ENTER_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.8. EXEC (0x08)

在设备上执行函数。该函数无参数且无返回值，必须通过RAM进行通信。执行此方法将阻塞其他命令及大容量存储接口U F2的写入操作，故应仅在独占模式下谨慎使用（且需按ARM EABI规范保存和恢复寄存器）。

此方法在常规（非IRQ）上下文中调用，且堆栈非常有限，因此函数应使用其自身堆栈。

表182。PICOBOOT
设备命令结构中
的执行函数

偏移量	名称	数值 / 描述
0x08	bCmdId	0x08 (EXEC)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	要执行的函数地址（系统会自动为您添加thumb位，以防您遗忘）。

2.8.5.4.9. VECTORIZE_FLASH (0x09)

请求将Mass Storage和PICOBOOT接口内部使用的闪存访问函数向量表复制至RAM，从而允许用自定义版本替换该方法的实现（例如，若开发板所用闪存不支持标准命令）。

表183。PICOBOOT
向量化闪存
命令结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x09 (VECTORIZE_FLASH)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	指向RAM中放置向量表位置的指针

闪存功能向量表

```
结构体 {
    uint32_t size; // 28
    uint32_t (*do_flash_enter_cmd_xip)();
    uint32_t (*do_flash_exit_xip)();
    uint32_t (*do_flash_erase_sector)();
    uint32_t (*do_flash_erase_range)(uint32_t addr, uint32_t size);
    uint32_t (*do_flash_page_program)(uint32_t addr, uint8_t *data);
    uint32_t (*do_flash_page_read)(uint32_t addr, uint8_t *data);
};
```

这些方法的签名和参数与bootrom中对应的闪存访问函数相同（见第2.8.3.1.3节）。

请注意，主机必须随后通过在RP2040上执行的 **EXEC**命令更新该表在RAM中的副本，否则主机通过PICOBOOT **WRITE**对该（现已激活的RAM中）向量表有重叠的任何RAM写入操作，都会导致复位并恢复使用默认的ROM闪存函数向量表。

2.8.5.5 控制请求

以下请求通过默认控制管道发送至该接口。

2.8.5.5.1. INTERFACE_RESET (0x41)

主机发送此控制请求以复位 PICOBOOT 接口。该命令执行以下操作：

- 清除所有大包端点上的 HALT 状态（如果已设置）
- 中止任何正在进行的 PICOBOOT 或大容量存储传输及任何闪存写入（此方法为终止卡死闪存传输的唯一途径）。
- 清除之前的命令结果
- 解除 EXCLUSIVE_ACCESS，并在因排他访问而弹出的情况下重新挂载大容量存储驱动器。

表 184. PICOBOOT
复位 PICOBOOT
接口控制

bmRequestType	bRequest	wValue	wIndex	wLength	数据
01000001b	01000001b	0000h	接口	0000h	无

该命令成功时返回空数据包作为响应。

2.8.5.5.2. GET_COMMAND_STATUS (0x42)

检索上一个命令（可能仍在处理中）的状态。PICOBOOT协议命令的成功完成通过批量传输管道确认，若操作仍在进行或失败（导致批量管道阻塞），则可通过此方法查询操作状态。

表185。PICOBOOT
获取上一个命令
状态控制

bmRequestType	bRequest	wValue	wIndex	wLength	数据
11000001b	01000010b	0000h	接口	0000h	无

该命令返回如下16字节响应数据

表186。PICOBOOT
获取上一个命令
状态控制
响应

偏移量	名称	描述	
0x00	dToken	命令中指定的用户令牌	
0x04	dStatusCode	OK (0)	命令已成功完成（或仍在执行中）
		UNKNOWN_CMD (1)	命令ID未被识别
		INVALID_CMD_LENGTH (2)	命令请求长度不正确
		INVALID_TRANSFER_LENGTH (3)	数据传输长度与命令不符
		INVALID_ADDRESS (4)	指定地址对该命令类型无效； 即地址类型与命令预期的Flash/RAM类型不符
		BAD_ALIGNMENT (5)	指定地址未根据命令要求正确对齐
		INTERLEAVED_WRITE (6)	大容量存储接口UF2写入操作干扰了当前操作。命令因状态未知而被放弃。注意：若您拥有排他访问权限，则不会发生此情况。
		REBOOTING (7)	设备正在重启，命令已被忽略。
0x08	bCmdId	命令的标识符	
	bInProgress	命令仍在执行时，值为1	否则为0
	保留	(6个字节的零值)	

2.9. 电源供应

RP2040 需要五个独立的电源。然而，在大多数应用中，其中几个电源可以合并并连接至单一电源。在典型应用中，通常只需单一3.3V电源。详见第2.9.7.1节，“单一3.3V电源”。

以下章节将介绍电源及多种潜在的电源方案。第5.6节，“电源”，包含详细的电源参数。

2.9.1. 数字IO供电 (IOVDD)

IOVDD 为芯片的数字IO供电，电压应为1.8V至3.3V的额定值。供电电压决定数字IO的外部信号电平，并应根据所需信号电平选择。详情请参阅第5.5.3节，“引脚规格”。所有数字输入输出(IO)共用相同电源，并在相同信号电平下运行。

应在每个芯片IOVDD引脚附近，通过100nF电容对IOVDD进行去耦。

⚠ 注意

如果数字IO以标称1.8V供电，应通过VOLTAGE_SELECT寄存器调整IO输入阈值。默认情况下，当数字IO供电电压处于2.5V至3.3V标称范围内时，IO输入阈值有效。详情见第2.19节“GPIO”。以1.8V电压供电且输入阈值设置为适用于2.5V至3.3V电源的模式为安全操作模式，但输入阈值不符合规范要求。以高于标称1.8V电压供电且输入阈值设置为1.8V电源，可能导致芯片损坏。

2.9.2. 数字核心供电 (DVDD)

DVDD为芯片核心数字逻辑供电，额定电压应为1.1V。芯片内置专用稳压器，允许DVDD由数字IO电源 (IOVDD) 或其他1.8V至3.3V额定电压的电源产生。芯片内稳压器输出引脚 (VREG_VOUT) 与DVDD供电引脚之间的连接位于芯片外部，因此DVDD在需要时可由芯片外部电源供电。

芯片每个DVDD引脚附近应并联一个100nF去耦电容。

2.9.3. 片上电压调节器输入电源 (VREG_VIN)

VREG_VIN为芯片内稳压器的输入电源。其额定输入电压应为1.8V至3.3V。为减少外部电源数量，VREG_VIN可与数字IO电源 (IOVDD) 共用。

应在靠近芯片 VREG_VIN 引脚处，于 VREG_VIN 与地之间连接一个 1 μ F 电容。

⚠ 注意

VREG_VIN 还为芯片的上电复位和欠压检测模块供电，因此即使未使用片内电压调节器，该引脚仍须供电。

有关片内电压调节器的详细信息，请参见第 2.10 节，“核心电源调节器”。

2.9.4. USB PHY供电 (USB_VDD)

USB_VDD 为芯片的 USB PHY 供电，应以标称 3.3V 电压供电。为减少外部电源数量，USB_VDD 可与数字 IO 电源 (IOVDD) 共用同一电源，前提是 IOVDD 也以 3.3V 供电。若 IOVDD 不是 3.3V 供电，则 USB PHY 需单独使用 3.3V 电源，详见第 2.9.7.3 节，“带有功能 USB 和 ADC 的 1.8V 数字 IO”。在 USB PHY 从不使用的应用中，USB_VDD 可连接至标称电压介于 1.8V 至 3.3V 之间的任一电源。示例请参见第 2.9.7.4 节，“单一 1.8V 电源”。USB_VDD 不应悬空未连接。

USB_VDD 应通过靠近芯片 USB_VDD 引脚的 100nF 电容进行去耦。

2.9.5. ADC供电 (ADC_AVDD)

ADC_AVDD 为芯片的模拟到数字转换器 (ADC) 供电。其标称供电电压范围为 1.8V 至 3.3V，但低于 2.97V 时，ADC 性能将受到影响。为减少外部电源数量，ADC_AVDD 可使用与数字 IO 供电 (IOVDD) 相同的电源。

① 注意

ADC_AVDD 供电电压可高于或低于 IOVDD，例如，可将 ADC 供电至 3.3V 以实现最佳性能，同时支持数字 IO 的 1.8V 信号电平。但 ADC 模拟输入端电压不得超过 IOVDD。例如，若 IOVDD 为 1.8V，则 ADC 输入端电压应限制在 1.8V。

高于 IOVDD 的电压将导致通过 ESD 保护二极管的泄漏电流。详见第 5.5.3 节，“引脚规格”。

应在芯片 ADC_AVDD 引脚附近通过一个 100nF 电容对 ADC_AVDD 进行去耦。

2.9.6. 电源顺序

RP2040 的电源可按任意顺序上电或断电。然而，若 ADC 电源 (ADC_AVDD) 在数字核心电源 (DVDD) 之前上电，或在数字核心电源断电后断电，ADC 电源可能出现短暂瞬态电流。此情况不会损害芯片，但可通过使 DVDD 在 ADC_AVDD 之前或同时上电，以及在 ADC_AVDD 之后或同时断电来避免。在最常见的电源方案中，芯片由单一 3.3V 电源供电时，因片内电压调节器的启动时间，DVDD 会在 ADC_AVDD 之后短暂延迟上电。此为可接受的行为。详见第 2.9.7.1 节，“单一 3.3V 电源”。

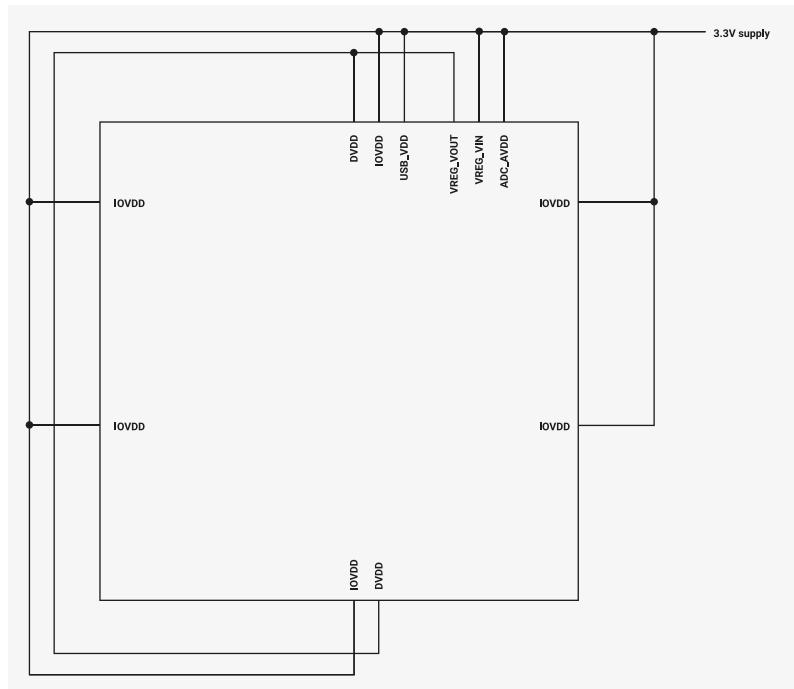
2.9.7. 电源方案

2.9.7.1. 单一 3.3V 电源

在大多数应用中，RP2040 将由单一 3.3V 电源供电，如图 16 所示。数字 IO (IOVDD)、USB PHY (USB_VDD) 及 ADC (ADC_AVDD) 将直接由 3.3V 电源供电，1.1V 数字核心电源 (DVDD) 则由片内电压调节器从 3.3V 电源调节得到。注意，调节器输出引脚 (VREG_VOUT) 必须连接至芯片片外的 DVDD 引脚。

有关片内电压调节器的详细信息，请参见第 2.10 节，“核心电源调节器”。

图16 采用单一3.3V电源为芯片供电
(简化示意图省略去耦元件)



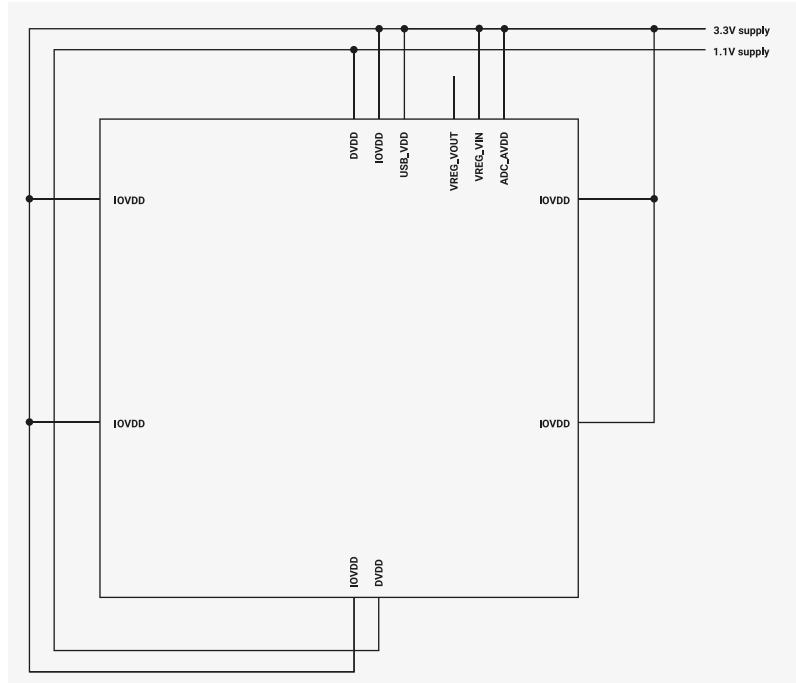
2.9.7.2. 外部核心供电

数字核心（DVDD）可直接由外部1.1V电源供电，而非来自片内调节器，如图17所示。若系统中存在合适的外部稳压器，或对于可用高效开关调节器取代低效线性片上稳压器的低功耗应用，该方案可能具有合理性。

若采用外部核心电源，则片上稳压器输出端（VREG_VOUT）应保持悬空。

但仍须向稳压器输入端（VREG_VIN）供电，以支持芯片的上电复位和欠压检测模块。片上稳压器在VREG_VIN可用时即自动上电，但芯片退出复位后可通过软件控制关闭。详见第2.10节，“核心电源稳压器”。

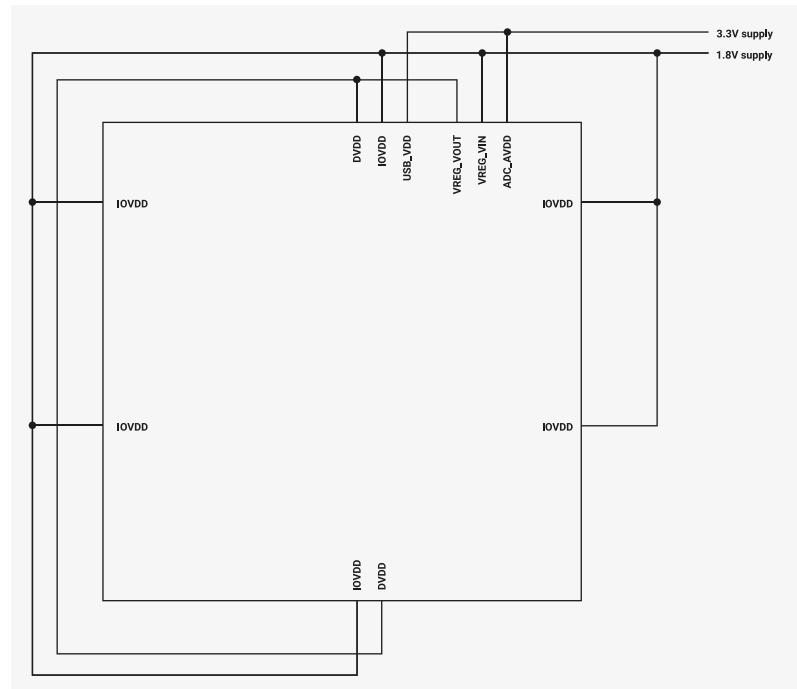
图17。使用外部核心电源



2.9.7.3. 具功能性USB和ADC的1.8V数字IO

数字IO信号电平低于3.3V的应用需为USB PHY和ADC提供独立3.3V电源，因USB PHY在低于3.135V时不符合规格，且ADC在低于2.97V时性能受影响。图18示例中，数字IO（IOVDD）供电为1.8V，USB PHY（USB_VDD）及ADC（ADC_AVDD）采用独立3.3V电源。在本示例中，稳压器输入端（VREG_VIN）连接至1.8V电源，亦可连接至3.3V电源。若1.8V电源由高效开关稳压器产生，连接至该电源可降低整体功耗。

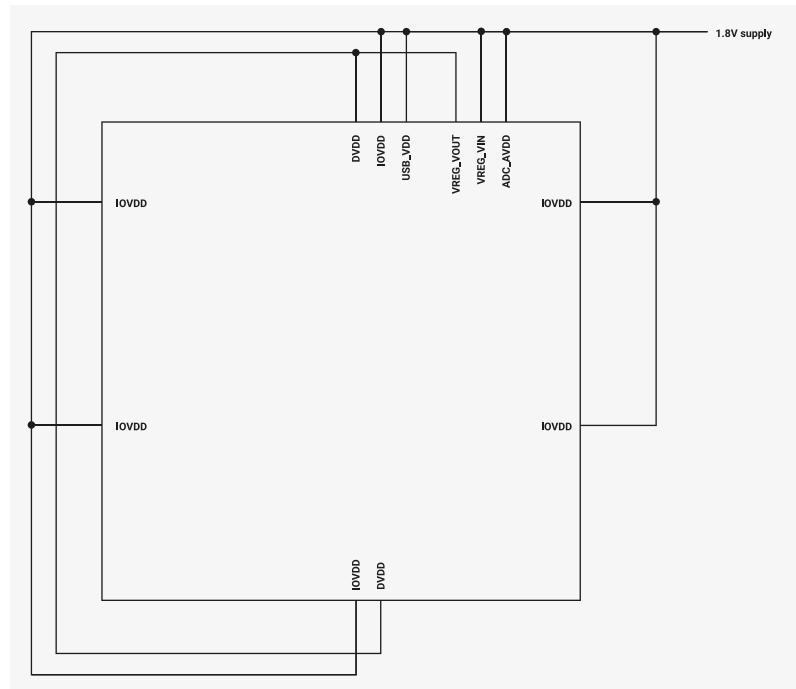
图18。在使用USB和ADC时支持1.8V IO



2.9.7.4 单一1.8V电源

若无需功能性USB PHY及最佳ADC性能，RP2040可由低于3.3V的单一电源供电。图19示例说明了单一1.8V电源的应用。在本示例中，核心电源（DVDD）由片上稳压器从1.8V电源调节获得。

图19。芯片由单
一1.8V电源供电



2.10. 核心供电调节器

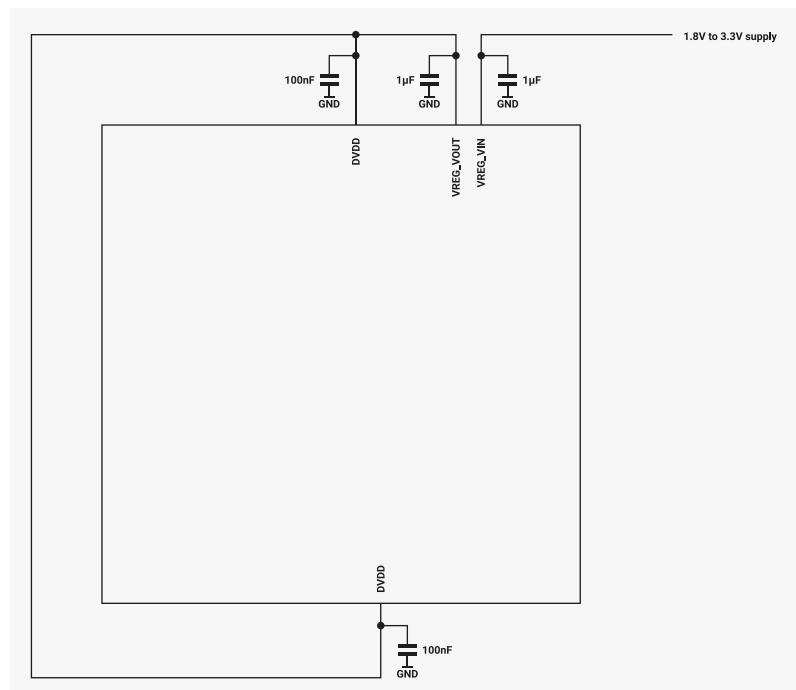
RP2040 包含片上电压调节器，允许数字核心电源（DVDD）由外部标称1.8V至3.3V电源生成。在大多数情况下，调节器的输入电源将与芯片的数字 IO 电源（IOVDD）共享外部电源，从而简化整体电源需求。

为使芯片能够启动，电压调节器默认启用，并在其输入电源可用时立即上电。芯片退出复位后，调节器可通过软件控制被禁用、置于高阻态或调整输出电压。输出电压可在0.80V至1.30V范围内以50mV步进设置，但初始上电或复位后，输出电压设定为标称1.1V。电压调节器最高可提供100mA电流。

虽然该电压调节器旨在为芯片数字核心电源（DVDD）供电，但若DVDD由外部电源直接供电，该调节器亦可用于其他用途。

2.10.1. 应用电路

图20。电压
调节器应用
电路



调节器的输入（VREG_VIN）和输出（VREG_VOUT）引脚附近必须放置 $1\mu F$ 电容。

2.10.2. 工作模式

电压调节器在三种模式之下工作。所选模式通过向VREG寄存器中的 EN和 HIZ字段写入实现，如表187所示。在初始上电或复位事件后，电压调节器处于正常工作模式。

表187。电压
调节器模式选择

模式	EN	HIZ
正常工作 ^a	1	0
高阻抗	1	1
关断	0	X

^a t 在初始上电或复位事件后，电压调节器处于正常模式

2.10.2.1. 正常工作模式

在正常运行模式下，电压调节器的输出电压保持在所选电压范围内，且调节器能够稳定供电。

2.10.2.2. 高阻抗模式

在高阻抗模式下，电压调节器被禁用，其输出引脚（VREG_VOUT）处于高阻抗状态。在该模式下，调节器的功耗降至最低。此模式允许连接到 VREG_VOUT 的负载由片内调节器以外的电源供电。例如，负载可先由片内电压调节器供电，然后通过软件控制切换至外部调节器供电。外部调节器亦须支持高阻抗模式，且任一时刻仅允许一个调节器为负载供电。当两者均处于高阻抗模式的短暂期间内，输出电容器维持调节器的供电电压。

2.10.2.3. 关闭模式

在关机模式下，电压调节器被禁用，功耗降至最低，调节器的输出引脚（VREG_VOUT）被拉至0V。

关机模式仅在电压调节器未为RP2040数字核心电源（DVDD）供电时才有效。如果调节器为DVDD供电且启用了欠压检测，进入关机模式将触发复位事件，电压调节器将恢复至正常模式。若未启用欠压检测，电压调节器将关闭，并保持在关机模式，直至其输入电源（VREG_VIN）断电重启。

2.10.3. 输出电压选择

所需输出电压可通过写入VREG寄存器中的 **VSEL** 字段进行选择。电压调节器的输出电压可在0.80V至1.30V范围内，以50mV为步进进行设置。调节器的输出电压在初次上电或复位后设置为1.1V。有关详细信息，请参阅VREG寄存器说明。

请注意，RP2040在其数字核心电源（DVDD）超出工作条件时可能无法稳定运行（参见第5.6节）；建议大多数应用中将稳压器设置为**1.10V**。

2.10.4. 状态

VREG寄存器包含一个状态字段，**ROK**，用以指示电压稳压器输出是否处于正确调节状态。

上电时，**ROK**保持低电平，直至稳压器启动且输出电压达到**ROK_{TH,ASSERT}**。随后，**ROK**保持高电平，直到电压降至**ROK_{TH,DEASSERT}**；之后保持低电平，直至输出电压再次超过断言阈值。**ROK_{TH,ASSERT}**的名义值为所选输出电压的90%，当输出电压为1.1V时，为0.99V；**ROK_{TH,DEASSERT}**的名义值为所选输出电压的87%，当输出电压为1.1V时，为0.957V。

请注意，将输出电压调高至更高电压时，**ROK**将保持低电平，直到达到更高电压的断言阈值。当稳压器处于高阻抗模式时，**ROK**亦将保持低电平。

2.10.5. 电流限制

该电压稳压器设有电流限制，以防止负载电流超过最大额定值。当电流限制生效时，输出电压将不受调节，且会低于所选值。

2.10.6. 寄存器列表

电压稳压器与芯片级复位子系统共享寄存器地址空间。两个子系统的寄存器列表见此处。仅VREG寄存器属于电压寄存器子系统。BOD和CHIP_RESET寄存器属于芯片级复位子系统。本文档其他部分将该共享地址空间称为**vreg_and_chip_reset**。

VREG_AND_CHIP_RESET寄存器起始地址为 **0x40064000**（在SDK中定义为VREG_AND_CHIP_RESET_BASE）。

表188。VREG_AND_CHIP_RESET寄存器列表

偏移量	名称	说明
0x0	VREG	电压调节器控制与状态
0x4	BOD	欠压检测控制
0x8	CHIP_RESET	芯片复位控制与状态

VREG_AND_CHIP_RESET: VREG寄存器

偏移: 0x0

描述

电压调节器控制与状态

表 189. VREG 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	ROK: 调节状态 0=未调节, 1=调节中	只读	0x0
11:8	保留。	-	-
7:4	VSEL: 输出电压选择 0000 至0101 - 0.80V 0110 - 0.85V 0111 - 0.90V 1000 - 0.95V 1001 - 1.00V 1010 - 1.05V 1011 - 1.10V (默认) 1100 - 1.15V 1101 - 1.20V 1110 - 1.25V 1111 - 1.30V	读写	0xb
3:2	保留。	-	-
1	HIZ: 高阻态模式选择 0=非高阻态模式, 1=高阻态模式	读写	0x0
0	EN: 使能 0=未使能, 1=已使能	读写	0x1

VREG_AND_CHIP_RESET: BOD 寄存器

偏移: 0x4

说明

欠压检测控制

表 190. BOD 寄存器

位	描述	类型	复位值
31:8	保留。	-	-

位	描述	类型	复位值
7:4	VSEL : 阈值选择 0000 - 0.473V 0001 - 0.516V 0010 - 0.559V 0011 - 0.602V 0100 - 0.645V 0101 - 0.688V 0110 - 0.731V 0111 - 0.774V 1000 - 0.817V 1001 - 0.860V (默认) 1010 - 0.903V 1011 - 0.946V 1100 - 0.989V 1101 - 1.032V 1110 - 1.075V 1111 - 1.118V	读写	0x9
3:1	保留。	-	-
0	EN : 使能 0=未使能, 1=已使能	读写	0x1

VREG_AND_CHIP_RESET: CHIP_RESET 寄存器

偏移: 0x8

描述

芯片复位控制与状态

表191。
CHIP_RESET 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	PSM_RESTART_FLAG : 此标志由调试器中的 psm_restart 设置。 其目的是在调试器执行 psm_restart 以从启动死锁中恢复时, 使启动代码跳转至安全模式。 在安全模式下, 调试器可修复启动代码, 清除此标志, 然后重新启动处理器。	WC	0x0
23:21	保留。	-	-
20	HAD_PSM_RESTART : 上次复位来源于调试端口	只读	0x0
19:17	保留。	-	-
16	HAD_RUN : 上次复位来源于RUN引脚	只读	0x0
15:9	保留。	-	-
8	HAD_POR : 上次复位来源于上电复位或欠压检测模块	只读	0x0
7:0	保留。	-	-

2.10.7. 详细规格

表192。电压
调节器详细
规格

参数	描述	最小值	典型值	最大值	单位
V_{VREG_VIN}	输入电源 电压	1.63	1.8 - 3.3	3.63	V
ΔV_{VREG_VOUT}	输出电压 变化	-3		+3	选定输出电压的 百分比
I_{MAX}	输出电流			100	mA
电流限制LIMIT	电流限制	150	350	450	mA
$ROK_{TH.ASSERT}$	ROK 断言 阈值	87	90	93	选定输出电压的 百分比
$ROK_{TH.DEASSERT}$	ROK 解除断言 阈值	84	87	90	选定输出电压的 百分比
$t_{\text{通电时间}}^a$	通电时间		275	350	μ秒

^a数值会因负载电流及VREG_VOUT上的电容而变化。条件：EN=1，负载电流=0mA，VREG_VIN在100μs内上升

2.11. 电源控制

RP2040提供多种降低动态功耗的选项：

- 对各单独外设及功能模块的顶层时钟门控
- 基于处理器睡眠状态的顶层时钟门控自动控制
- 系统时钟频率或时钟源的动态切换（例如切换至内部环振荡器，禁用PLL和晶振）
- 零动态功耗的休眠（DORMANT）状态，可由GPIO事件或RTC中断唤醒

RP2040上的所有数字逻辑均处于单一核心电源域。以下选项可用于降低静态功耗：

- 将存储器置于保持状态的断电模式
- 对支持此功能的外围设备执行电源门控，例如ADC和温度传感器

2.11.1. 顶层时钟门控

每个时钟域（例如系统时钟）可能驱动大量不同的硬件模块，且不一定同时全部需要。为避免不必要的功率消耗，每个时钟的单独终端（例如UART系统时钟输入）可以随时被禁用。

启用和禁用时钟门控操作无毛刺。若外围设备时钟被暂时禁用，随后重新启用，外围设备将保持禁用前的状态。无需复位或重新初始化。

时钟门控由两组寄存器控制：WAKE_ENx寄存器（自WAKE_EN0起）和SLEEP_ENx寄存器（自SLEEP_EN0起）。这两组寄存器位级完全相同，每个位均含控制相应时钟终端的标志位。WAKE_EN寄存器指定系统处于唤醒状态时启用的时钟，SLEEP_ENx寄存器则选择处理器处于SLEEP状态时启用的时钟（参见第2.11.2节）。

两个Cortex-M0+处理器不具有外部可控的时钟门控。处理器会基于执行的WFI/WFE指令以及外部事件（Event）和中断请求（IRQ）信号，自主控制其子系统的时钟门控。

2.11.2. SLEEP 状态

当满足以下所有条件时，RP2040进入SLEEP状态：

- 两个处理器均处于休眠状态（例如执行 `WFE` 或 `WFI` 指令）
- 系统DMA在任一通道上均无未完成的传输

当任一处理器被中断唤醒时，RP2040即退出SLEEP状态。

处于SLEEP状态时，顶级时钟门控由SLEEP_ENx寄存器（从SLEEP_EN0开始）进行屏蔽，而非WAKE_ENx寄存器。此功能允许在处理器睡眠时对时钟树进行更严格的修剪。

注意

虽然时钟可以在SLEEP期间启用而在非SLEEP期间禁用，但此做法通常无实用价值。

例如，若系统处于睡眠状态直至UART字符中断，除UART外的整个系统均可进行时钟门控（SLEEP_ENx 除CLK_SYS_UART0和CLK_PERI_UART0外全为零），其中包括总线结构等系统基础设施。

当UART触发中断并唤醒处理器时，RP2040退出SLEEP模式，并切换回WAKE_ENx时钟掩码。至少应包含总线结构及存储处理器堆栈和中断向量的存储设备。

系统级时钟请求握手机制会在时钟重新启用前，阻止处理器访问总线。

2.11.3. 休眠状态

DORMANT状态为真正的零动态功耗睡眠状态，所有时钟及振荡器均被禁用。系统可通过GPIO事件（高/低电平或上升/下降沿）或RTC中断从DORMANT状态唤醒：该操作重新启动其中一个振荡器（环形振荡器或晶体振荡器），并在振荡器输出稳定后解除门控。系统状态被保留，代码执行将在离开DORMANT状态后立即恢复。

注意，若依赖RTC（第4.8节）从DORMANT状态唤醒，RTC必须具有外部时钟源。RTC接受低至1Hz的时钟频率。

还需注意，DORMANT状态并不会停止PLL。为避免不必要的功耗，软件应在进入DORMANT状态前关闭PLL，并在退出后重新上电及重新配置PLL。

通过向当前活动振荡器的DORMANT寄存器写入关键字进入DORMANT状态：环形振荡器（第2.17节）或晶体振荡器（第2.16节）。若两者均处于活动状态，必须最后停止为处理器提供时钟的振荡器，否则将导致软件停止执行。

2.11.4. 内存断电

主系统存储器（SRAM0...5，映射至总线地址 `0x20000000` 至 `0x20041fff`），以及USB DPRAM，可通过 Syscfg 寄存器中的 MEMPOWERDOWN 寄存器断电（详见第 2.21 节）。断电时，存储器保持当前内容，但无法被访问。静态功耗得以降低。

⚠ 注意

断电状态下不得访问存储器。否则可能导致存储器内容损坏。

重新上电存储器后，需延迟 20ns 后方可再次访问存储器。

XIP 缓存（详见第 2.6.3 节）亦可通过 CTRL.POWER_DOWN 实现断电。缓存断电期间，XIP 硬件不会产生缓存访问请求。注意，若代码仍从 XIP 执行，鉴于外部 QSPI 总线较高的电压和开关电容，通常无法实现净功耗节约。

2.11.5. 程序员模型

2.11.5.1 睡眠

示例hello_sleep位于https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c，演示了睡眠模式。hello_sleep 应用程序（及其底层函数）执行以下步骤：

- 将系统中所有时钟设为由 XOSC 驱动
- 在 RTC 中配置一个 10 秒后的闹钟
- 使用 SLEEP_ENx 寄存器（参见 SLEEP_EN0）设置 clk_RTC 为睡眠模式下唯一运行的时钟
- 启用处理器深度睡眠
- 在处理器上调用 `_wfi` 指令，使处理器进入深度睡眠状态，直至被 RTC 中断唤醒
- RTC 中断清除闹钟后调用用户提供的回调函数
- 回调函数结束示例应用程序

 ⓘ 注意

须在 proc0 和 proc1 上均启用深度睡眠并调用 `_wfi`，且确保 DMA 停止，方可进入睡眠模式。

hello_sleep 利用 Pico Extras 中 `pico_sleep` 的函数。特别是，`sleep_goto_sleep_until` 使处理器进入睡眠状态，直至被假定在未来的 RTC 时间唤醒。

Pico 附加功能：https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第159至183行

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // 我们应已调用过 sleep_run_from_dormant_source 函数
163 // 该功能仅在休眠状态下需要，尽管它在睡眠时通过 xosc 节约了功耗
164
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCKS_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;
169
170     aon_timer_enable_alarm(ts, callback, false);
171
172     stdio_flush();
173
174     // 在处理器上启用深度睡眠
175     processor_deep_sleep();

```

```

175
176     // 进入睡眠状态
177     __wfi();
178 }

```

2.11.5.2. 休眠模式

hello_dormant 示例，https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_dormant/hello_dormant_gpio.c，演示了休眠模式。该示例执行以下步骤：

- 将系统中所有时钟设为由 XOSC 驱动
- 为“dormant_wake”硬件配置 GPIO 中断，此硬件可唤醒处于休眠模式的 ROSC 和 XOSC
- 将 XOSC 置于休眠模式，该模式停止所有处理器执行（以及芯片上所有其他时钟逻辑）
立即
- 当 GPIO 10 变为高电平时，XOSC 重新启动，程序执行继续

hello_dormant 底层使用 sleep_goto_dormant_until_pin 函数：

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第258至282行

```

258 void sleep_goto_dormant_until_pin(uint gpio_pin, bool edge, bool high) {
259     bool low = !high;
260     bool level = !edge;
261
262     // 在 IO bank 0 配置相应的中断请求 (IRQ)
263     assert(gpio_pin < NUM_BANK0_GPIOS);
264
265     uint32_t event = 0;
266
267     if (level && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_LOW_BITS;
268     if (level && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_HIGH_BITS;
269     if (edge && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_HIGH_BITS;
270     if (edge && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_LOW_BITS;
271
272     gpio_init(gpio_pin);
273     gpio_set_input_enabled(gpio_pin, true);
274     gpio_set_dormant_irq_enabled(gpio_pin, event, true);
275
276     _go_dormant();
277     // 执行在此处停止，直到被唤醒
278
279     // 清除中断请求，以便我们如有需要可再次进入休眠模式
280     gpio_acknowledge_irq(gpio_pin, event);
281     gpio_set_input_enabled(gpio_pin, false);
282 }

```

2.12. 芯片级复位

2.12.1. 概述

芯片级复位子系统复位整个芯片，使其恢复到默认状态。此操作发生在初次上电时、电源欠压事件期间，或芯片的RUN引脚被拉低时。芯片也可以通过

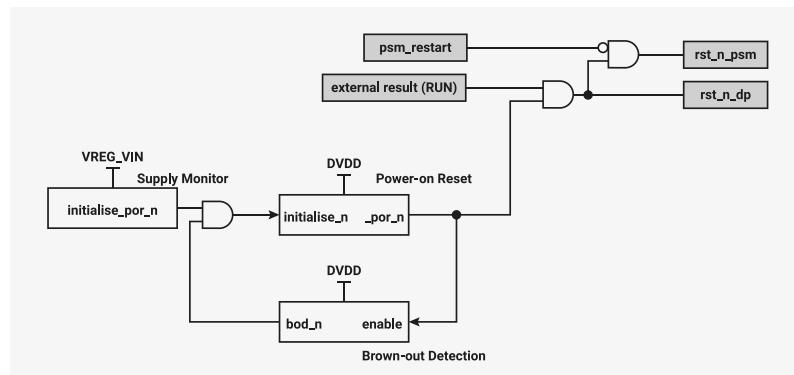
救援调试端口复位。详情请参见第2.3.4.2节，“救援DP”。

该子系统具有两个复位输出。`rst_n_psm`用以复位整个芯片（调试端口除外），`rst_n_dp`仅复位救援DP。两者在初次上电、电源欠压或RUN引脚为低电平时均保持低电平状态。

`rst_n_psm`还可以通过子系统的`psm_restart`输入，由救援DP保持为低电平。该功能允许通过Rescue DP复位芯片，而不复位Rescue DP本身。子系统通过将`rst_n_psm`拉高以释放芯片复位，进而移交控制权给上电状态机，后者继续启动芯片。详情请参见第2.13节，“上电状态机”。

芯片级复位子系统如图21所示，相关信息请见后续章节。

图21。芯片级复位子系统

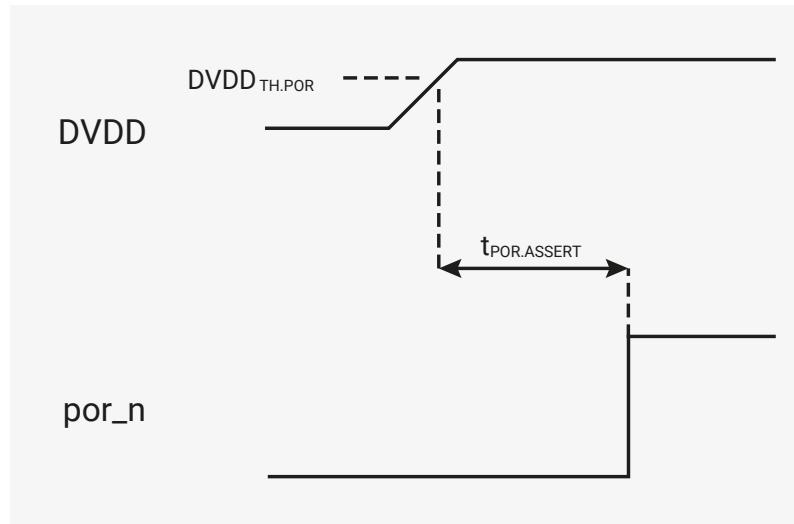


2.12.2. 上电复位

上电复位模块通过保持芯片处于复位状态，直到数字核心电源（DVDD）能够稳定供电给芯片核心逻辑，确保芯片首次上电时的正常启动。该模块将其`por_n`输出保持低电平，直至DVDD高于上电复位阈值（DVDD TH.POR）连续持续超过上电复位保持延时（ $t_{POR ASSERT}$ ）。一旦`por_n`输出变为高电平，即使DVDD随后降至低于DVDD TH.POR，`por_n`仍保持高电平，除非启用了欠压检测功能。加电时`por_n`的行为如图22所示。

DVDD_{TH.POR}固定为标称0.957V，阈值应介于0.924V与0.99V之间。该阈值假定初次加电时DVDD的标称值为1.1V，若使用较低电压，`por_n`可能永远不会置高。芯片解除复位后，只要禁用欠压检测或已设置适当的阈值电压，DVDD可降低而不会导致`por_n`置低。

图22. 上电复位周期



2.12.2.1. 详细规格

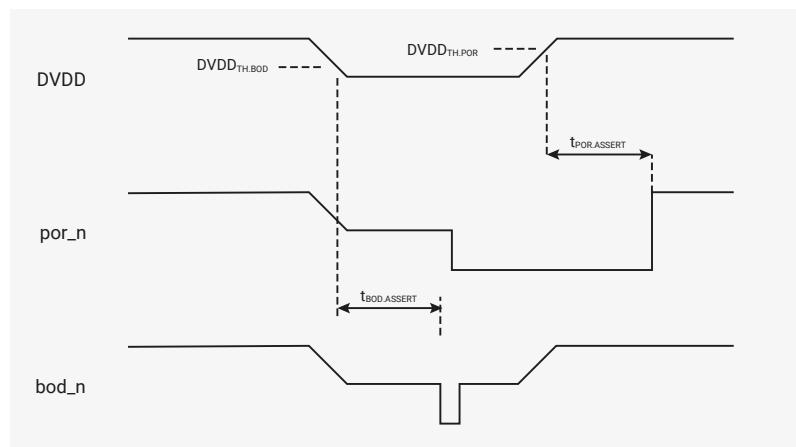
表193。上电复位参数

参数	描述	最小值	典型值	最大值	单位
DVDD _{TH.POR}	上电复位阈值	0.924	0.957	0.99	V
t _{POR ASSERT}	上电复位断言延迟		3	10	μ秒

2.12.3. 欠压检测

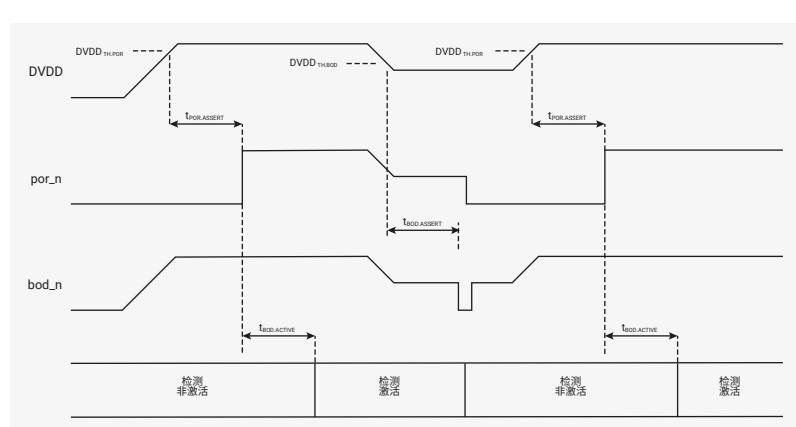
欠压检测模块通过在数字核供电电压（DVDD）降至安全工作电平以下时启动加电复位周期，防止不可靠操作。当DVDD降至棕色欠压检测阈值（DVDD_{TH.BOD}）以下且持续时间超过棕色欠压检测断言延迟（t_{BOD ASSERT}）时，模块的bod_n输出将被置低。此操作重新初始化上电复位模块，该模块通过将其por_n输出置低来复位芯片，并持续保持复位状态，直到DVDD恢复至安全工作电平。图23显示了一个棕色欠压事件及其后续的上电复位周期。

图23。一次棕色欠压检测周期



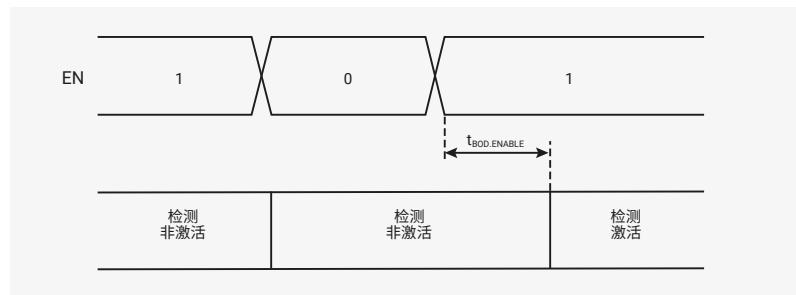
棕色欠压检测在初次上电或因棕色欠压引发的复位后自动启用。然而，在por_n返回高电平时与检测启动之间存在短暂延时，即棕色欠压检测激活延迟（t_{BOD ACTIVE}）。如图24所示。

图24。上电初始化及棕褐断电事件后棕褐断电检测的激活。



通过向BOD寄存器的EN字段写入0可禁用检测，写入1则可重新启用检测。检测禁用时模块的bod_n输出为高电平。

图25。棕褐断电检测的禁用与启用



BOD寄存器复位时检测将重新启用，因为这会将寄存器的EN字段设置为1。经过等同于棕褐断电检测启用延迟($t_{BOD.ENABLE}$)的延时后，检测将变为激活状态。

① 注意

如果BOD寄存器因上电复位或棕褐断电复位而复位，则从寄存器复位到棕褐断电检测激活之间的延迟等于棕褐断电检测激活延迟($t_{BOD.ACTIVE}$)。对于所有其他复位源，延迟等于棕褐断电检测使能延迟($t_{BOD.ENABLE}$)。

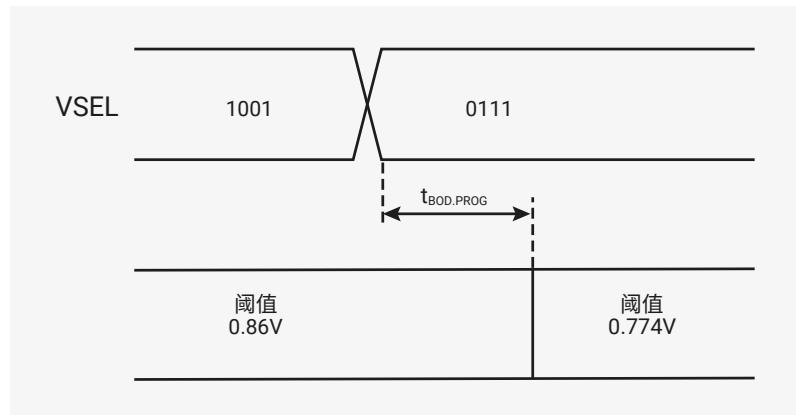
2.12.3.2. 调整检测阈值

棕褐断电检测阈值(DVDDTH.BOD)在初次上电或复位事件后，标称值为0.86V。

此阈值应在0.83V到0.89V之间。复位结束后，阈值可由软件控制进行调整。新的检测阈值将在棕褐断电检测编程延迟($t_{BOD.PROG}$)后生效。图26展示了相关示例。

阈值通过向BOD寄存器中的VSEL字段写入数据来调整。详见BOD寄存器说明。

图26。调整欠压检测阈值



2.12.3.3. 详细规格

表194。欠压检测参数

参数	描述	最小值	典型值	最大值	单位
$DVDD_{TH.BOD}$	欠压检测阈值	96.5	100	103.5	所选阈值电压的百分比
$t_{BOD.ACTIVE}$	欠压检测激活延迟		55	80	μ 秒

参数	描述	最小值	典型值	最大值	单位
$t_{BOD ASSERT}$	欠压 检测 断言延迟		3	10	μ秒
$t_{BOD ENABLE}$	欠压 检测启用 延迟		35	55	μ秒
$t_{BOD PROG}$	欠压 检测 编程 延迟		20	30	μ秒

2.12.4. 电源监控

上电和欠压复位模块由片内电压调节器的输入电源（VREG_VIN）供电。模块在首次加电时初始化，但若VREG_VIN尚未降至足够低电平即断电并重新加电，模块可能无法可靠地重新初始化。为防止该情况发生，需监控VREG_VIN电压，当其降至VREG_VIN激活阈值（VREG_VIN_{TH ACTIVE}）以下时，将重新初始化上电复位模块。

VREG_VIN_{TH ACTIVE}固定为名义值1.1V，该阈值范围应介于0.87V至1.26V之间。该阈值并非安全工作电压。该电压为VREG_VIN必须低于的值，以保证可靠地重新初始化上电复位模块。为确保安全运行，VREG_VIN的名义电压须保持在1.8V至3.3V之间。

2.12.4.1. 详细规格

表195。电压调节器
输入电源
监测参数

参数	描述	最小值	典型值	最大值	单位
VREG_VIN _{TH ACTIVE}	VREG_VIN 激活 阈值	0.87	1.1	1.26	V

2.12.5. 外部复位

芯片亦可通过将RUN引脚拉低进行复位。无论核心电源（DVDD）及上电复位/欠压检测模块状态如何，拉低RUN均可使芯片保持复位状态。一旦RUN引脚拉高，且所有其他复位源均已释放，芯片即会退出复位状态。RUN引脚可用于延长初始上电复位时间，或由外部信号驱动，以按需启动和停止芯片。若不使用RUN引脚，应将其固定为高电平。

2.12.6. 救援调试端口复位

芯片亦可通过救援调试端口复位，从而实现从死锁状态恢复。救援调试端口复位不仅复位芯片，同时还会在CHIP_RESET寄存器中设置PSM_RESTART_FLAG标志位。启动时，bootcode会检查该标志位，若被置位则进入安全状态。详情请参见第2.3.4.2节“Rescue DP”。

2.12.7. 最近一次复位来源

最近一次芯片级复位的来源通过读取CHIP_RESET寄存器中 HAD_POR、HAD_RUN和HAD_PSM_RESTART字段状态可得知。HAD_POR字段为1表示电源相关复位，即上电复位或欠压复位；HAD_RUN字段为1表示芯片最近一次复位由RUN引脚触发；HAD_PSM_RESTART字段为1表示芯片通过救援调试端口（Rescue Debug Port）复位。三个字段中不应有多个同时为1。

2.12.8. 寄存器列表

芯片级复位子系统与片上电压调节器共享寄存器地址空间。这两个子系统的寄存器均列于第2.10.6节。文档其他部分将该共享地址空间称为vreg_and_chip_reset。

2.13. 上电状态机

2.13.1. 概述

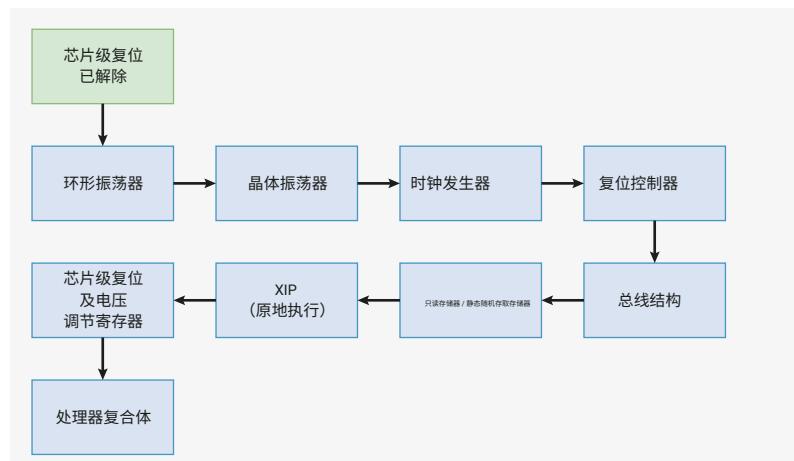
上电状态机按照特定顺序依次解除各硬件模块的复位信号。上电状态机中每个外设均由内部低电平有效复位信号rst_n控制，并产生内部高电平有效复位完成信号rst_done。上电状态机先解除当前外设的复位，等待该外设断言其rst_done信号后，方可解除下一个外设的复位。此机制确保芯片内时钟源稳定运行后，才能解除对时钟发生器的复位。此措施避免了将可能发生故障的时钟分发至芯片。

当芯片级复位子系统确认数字核心电源（DVDD）供电稳定且RUN引脚为高电平时，上电状态机解除复位。此时，上电状态机通过其rst_n_run输出将多个其他模块解除复位。该输出用于复位启动时需复位但在上电状态机重启时不得复位的模块，具体包括：

- 环形振荡器和晶体振荡器中的上电逻辑
- 在上电状态机重启期间必须持续运行的时钟分频器（clk_ref和clk_sys）
- 看门狗（包含需在上电状态机软重启期间保持的暂存寄存器）

2.13.2. 上电序列

图 27. 上电
状态机
序列。



上电状态机序列如下：

- 当数字核心电源（DVDD）通电且稳定，且 `RUN`引脚保持高电平（此时 `rst_n_run` 同样解除断言）时，芯片级复位子系统解除上电状态机的复位。
- 启动环形振荡器。当环形计数器检测到足够数量的时钟边沿，表明环形振荡器稳定时，`rst_done`被断言。
- 晶体振荡器复位信号被解除。此时晶体振荡器尚未启动，因此 `rst_done`立即被断言。
- 解除 `clk_ref` 和 `clk_sys` 时钟发生器的复位。在初始配置中，`clk_ref` 由无分频的环形振荡器直接驱动。`clk_sys` 由 `clk_ref` 驱动。这些时钟对于后续序列的执行是必需的。

序列的其余部分较为简洁，复位后以下模块将依次启动：

- 复位控制器——用于复位所有非引导外设
- 芯片级复位及电压调节器寄存器——由引导ROM用于检测芯片的引导状态。特别地，CHIP_RESET寄存器中的`SM_RESTART_FLAG`标志可通过SWD设置，以告知引导代码闪存中存在错误代码，需停止执行。CHIP_RESET寄存器的复位状态由芯片级复位子系统决定，且不受上电状态机复位的影响。
- XIP（Execute-In-Place，原地执行）——引导ROM用于从外部SPI闪存执行代码
- ROM和SRAM——引导代码从ROM执行，SRAM供处理器和总线结构使用。
- 总线结构——使处理器能够与外设通信
- 处理器复合体——处理器最终开始运行

复位结束时，最后启动的是处理器复合体。这包括 `core0` 和 `core1`。两个核心都会开始从ROM执行启动代码。启动ROM执行的首要步骤之一是读取核心ID。此时，`core1` 将进入休眠状态，由 `core0` 继续执行启动ROM。处理器复合体具有自身的复位控制和多种低功耗模式，这就是为什么即使仅需 `core0` 执行启动ROM，`core0` 和 `core1` 的复位仍被解除断言。

2.13.3. 寄存器控制

加电状态机是一种完全自动化的硬件装置。其运行无需用户输入。设有寄存器控制，可用于覆盖并监测加电状态机的状态。如有必要，可通过软件重置加电状态机中的硬件模块。另有一个 `WDSEL` 寄存器，用于控制看门狗复位时被复位的内容。

2.13.4. 与看门狗的交互

当看门狗触发时，可从软件可编程的位置重新启动加电状态机。例如，在处理器陷入无限循环，或程序员错误配置芯片的情况下。

须注意，若电源开启状态机中的某外设设置了 `WDSEL` 位，则电源开启序列中其后续的所有外设亦将被复位，因所选外设的 `rst_done` 信号将被撤销，进而对剩余外设施加复位信号 `rst_n`。

2.13.5. 寄存器列表

PSM寄存器的基地址为 `0x40010000`（在SDK中定义为`PSM_BASE`）。

表196. PSM
寄存器列表

偏移量	名称	说明
0x0	FRCE_ON	强制解除复位（即上电）
0x4	FRCE_OFF	强制进入复位（即断电）
0x8	WDSEL	若该外设在看门狗触发时需复位，则此位设为1。
0xc	DONE	指示该外设寄存器已准备好访问。

PSM: FRCE_ON 寄存器

偏移: 0x0

描述

强制解除复位（即上电）

表197. FRCE_ON
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: FRCE_OFF 寄存器

偏移: 0x4

说明

强制进入复位（即断电）

表198. FRCE_OFF
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0

位	描述	类型	复位值
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: WDSEL 寄存器

偏移: 0x8

描述

若该外设在看门狗触发时需复位，则此位设为1。

表199. WDSEL
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0

位	描述	类型	复位值
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: DONE 寄存器

偏移: 0xc

描述

指示该外设寄存器已准备好访问。

表 200. 完成寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	只读	0x0
15	PROC0	只读	0x0
14	SIO	只读	0x0
13	VREG_AND_CHIP_RESET	只读	0x0
12	XIP	只读	0x0
11	SRAM5	只读	0x0
10	SRAM4	只读	0x0
9	SRAM3	只读	0x0
8	SRAM2	只读	0x0
7	SRAM1	只读	0x0
6	SRAM0	只读	0x0
5	ROM	只读	0x0
4	BUSFABRIC	只读	0x0
3	RESETS	只读	0x0
2	CLOCKS	只读	0x0
1	XOSC	只读	0x0
0	ROSC	只读	0x0

2.14. 子系统复位

2.14.1. 概述

复位控制器允许软件控制对 RP2040 中所有非关键启动外设的复位，包括：

- USB 控制器

- PIO
- UART、I2C、SPI、PWM、定时器、ADC 等外设
- 锁相环（PLL）
- IO 与引脚寄存器

完整列表详见寄存器说明。

由复位控制器复位的每个外设均在上电时保持复位状态。软件需负责取消其计划使用的外设的复位。请注意，使用 SDK 时，部分外设可能已解除复位。

2.14.2. 程序员模型

SDK 定义了表示复位寄存器的结构体。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/resets.h 第 59 至 146 行

```

59 typedef struct {
60     _REG_(RESETS_RESET_OFFSET) // RESETS_RESET
61     //重置控制。
62     // 0x01000000 [24]    USBCTRL      (1)
63     // 0x00800000 [23]    UART1        (1)
64     // 0x00400000 [22]    UART0        (1)
65     // 0x00200000 [21]    TIMER         (1)
66     // 0x00100000 [20]    TBMAN         (1)
67     // 0x00080000 [19]    SYSINFO       (1)
68     // 0x00040000 [18]    SYSCFG        (1)
69     // 0x00020000 [17]    SPI1          (1)
70     // 0x00010000 [16]    SPI0          (1)
71     // 0x00008000 [15]    RTC           (1)
72     // 0x00004000 [14]    PWM           (1)
73     // 0x00002000 [13]    PLL_USB        (1)
74     // 0x00001000 [12]    PLL_SYS        (1)
75     // 0x00000800 [11]    PIO1          (1)
76     // 0x00000400 [10]    PIO0          (1)
77     // 0x00000200 [9]     PADS_QSPI     (1)
78     // 0x00000100 [8]     PADS_BANK0    (1)
79     // 0x00000080 [7]     JTAG           (1)
80     // 0x00000040 [6]     IO_QSPI       (1)
81     // 0x00000020 [5]     IO_BANK0      (1)
82     // 0x00000010 [4]     I2C1          (1)
83     // 0x00000008 [3]     I2C0          (1)
84     // 0x00000004 [2]     DMA            (1)
85     // 0x00000002 [1]     BUSCTRL       (1)
86     // 0x00000001 [0]     ADC            (1)
87     io_rw_32 reset;
88
89     _REG_(RESETS_WDSEL_OFFSET) // RESETS_WDSEL
90     //看门狗选择
91     // 0x01000000 [24]    USBCTRL      (0)
92     // 0x00800000 [23]    UART1        (0)
93     // 0x00400000 [22]    UART0        (0)
94     // 0x00200000 [21]    TIMER         (0)
95     // 0x00100000 [20]    TBMAN         (0)
96     // 0x00080000 [19]    SYSINFO       (0)
97     // 0x00040000 [18]    SYSCFG        (0)
98     // 0x00020000 [17]    SPI1          (0)
99     // 0x00010000 [16]    SPI0          (0)
100    // 0x00008000 [15]    RTC           (0)
101    // 0x00004000 [14]    PWM           (0)
102    // 0x00002000 [13]    PLL_USB        (0)

```

```

103 // 0x00001000 [12]    PLL_SYS      (0)
104 // 0x00000800 [11]    PIO1        (0)
105 // 0x00000400 [10]    PIO0        (0)
106 // 0x00000200 [9]     PADS_QSPI   (0)
107 // 0x00000100 [8]     PADS_BANK0 (0)
108 // 0x00000080 [7]     JTAG         (0)
109 // 0x00000040 [6]     IO_QSPI    (0)
110 // 0x00000020 [5]     IO_BANK0  (0)
111 // 0x00000010 [4]     I2C1        (0)
112 // 0x00000008 [3]     I2C0        (0)
113 // 0x00000004 [2]     DMA          (0)
114 // 0x00000002 [1]     BUSCTRL    (0)
115 // 0x00000001 [0]     ADC          (0)
116 io_rw_32 wdsel;
117
118 _REG_(RESETS_RESET_DONE_OFFSET) // RESETS_RESET_DONE
119 // 复位完成。
120 // 0x01000000 [24]    USBCTRL    (0)
121 // 0x00800000 [23]    UART1      (0)
122 // 0x00400000 [22]    UART0      (0)
123 // 0x00200000 [21]    TIMER       (0)
124 // 0x00100000 [20]    TBMAN      (0)
125 // 0x00080000 [19]    SYSINFO    (0)
126 // 0x00040000 [18]    SYSCFG     (0)
127 // 0x00020000 [17]    SPI1        (0)
128 // 0x00010000 [16]    SPI0        (0)
129 // 0x00008000 [15]    RTC         (0)
130 // 0x00004000 [14]    PWM         (0)
131 // 0x00002000 [13]    PLL_USB    (0)
132 // 0x00001000 [12]    PLL_SYS    (0)
133 // 0x00000800 [11]    PIO1        (0)
134 // 0x00000400 [10]    PIO0        (0)
135 // 0x00000200 [9]     PADS_QSPI  (0)
136 // 0x00000100 [8]     PADS_BANK0 (0)
137 // 0x00000080 [7]     JTAG         (0)
138 // 0x00000040 [6]     IO_QSPI    (0)
139 // 0x00000020 [5]     IO_BANK0  (0)
140 // 0x00000010 [4]     I2C1        (0)
141 // 0x00000008 [3]     I2C0        (0)
142 // 0x00000004 [2]     DMA          (0)
143 // 0x00000002 [1]     BUSCTRL    (0)
144 // 0x00000001 [0]     ADC          (0)
145 io_ro_32 reset_done;
146 } resets_hw_t;

```

定义了三个寄存器：

- **reset**: 该寄存器包含针对每个可复位外设的位。如果该位被设置为 1，则复位被激活。
如果该位被清除，则复位被解除。
- **wdsel**: 若设置该位，则在看门狗触发时，该外设将被复位（注意，电源上电状态机可能复位整个复位控制器，从而复位所有外设）。
- **reset_done**: 每个外设对应一个位，外设完成复位后该位被置位。这允许软件等待该状态位，以确保外设完成初始化后方可使用。

SDK 中复位功能定义如下：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第121至123行

```

121 static __force_inline void reset_block(uint32_t bits) {
122     reset_block_mask(bits);

```

```
123 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第125至127行

```
125 static __force_inline void unreset_block(uint32_t bits) {
126     unreset_block_mask(bits);
127 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第129至131行

```
129 static __force_inline void unreset_block_wait(uint32_t bits) {
130     return unreset_block_mask_wait_blocking(bits);
131 }
```

这些用法的一个示例出现在UART驱动中，驱动根据指定的uart选择复位寄存器的不同位，定义了uart_reset函数：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c 第32至38行

```
32 static inline void uart_reset(uart_inst_t *uart) {
33     reset_block_num(uart_get_reset_num(uart));
34 }
35
36 static inline void uart_unreset(uart_inst_t *uart) {
37     unreset_block_num_wait_blocking(uart_get_reset_num(uart));
38 }
```

2.14.3. 寄存器列表

复位控制器寄存器基地址为 **0x4000c000**（在SDK中定义为RESETS_BASE）。

表201.
RESETS寄存器列表

偏移量	名称	说明
0x0	RESET	复位控制。
0x4	WDSEL	看门狗选择。
0x8	RESET_DONE	复位完成。

RESETS：RESET寄存器

偏移: 0x0

描述

复位控制。位被置位表示外围设备处于复位状态；0表示外围设备的复位已解除。

表202. RESET
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	读写	0x1
23	UART1	读写	0x1
22	UART0	读写	0x1
21	TIMER	读写	0x1

位	描述	类型	复位值
20	TBMAN	读写	0x1
19	SYSINFO	读写	0x1
18	SYSCFG	读写	0x1
17	SPI1	读写	0x1
16	SPI0	读写	0x1
15	RTC	读写	0x1
14	PWM	读写	0x1
13	PLL_USB	读写	0x1
12	PLL_SYS	读写	0x1
11	PIO1	读写	0x1
10	PIO0	读写	0x1
9	PADS_QSPI	读写	0x1
8	PADS_BANK0	读写	0x1
7	JTAG	读写	0x1
6	IO_QSPI	读写	0x1
5	IO_BANK0	读写	0x1
4	I2C1	读写	0x1
3	I2C0	读写	0x1
2	DMA	读写	0x1
1	BUSCTRL	读写	0x1
0	ADC	读写	0x1

复位：WDSEL寄存器

偏移：0x4

说明

看门狗选择。若某位被置位，则当看门狗触发时，该看门狗将重置对应外设。

表 203. WDSEL
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	读写	0x0
23	UART1	读写	0x0
22	UART0	读写	0x0
21	TIMER	读写	0x0
20	TBMAN	读写	0x0
19	SYSINFO	读写	0x0
18	SYSCFG	读写	0x0
17	SPI1	读写	0x0

位	描述	类型	复位值
16	SPI0	读写	0x0
15	RTC	读写	0x0
14	PWM	读写	0x0
13	PLL_USB	读写	0x0
12	PLL_SYS	读写	0x0
11	PIO1	读写	0x0
10	PIO0	读写	0x0
9	PADS_QSPI	读写	0x0
8	PADS_BANK0	读写	0x0
7	JTAG	读写	0x0
6	IO_QSPI	读写	0x0
5	IO_BANK0	读写	0x0
4	I2C1	读写	0x0
3	I2C0	读写	0x0
2	DMA	读写	0x0
1	BUSCTRL	读写	0x0
0	ADC	读写	0x0

重置：RESET_DONE 寄存器

偏移: 0x8

描述

重置完成。若某位被置位，表明外设已返回重置完成信号，指示外设寄存器已准备好访问。

表 204。
RESET_DONE 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	只读	0x0
23	UART1	只读	0x0
22	UART0	只读	0x0
21	TIMER	只读	0x0
20	TBMAN	只读	0x0
19	SYSINFO	只读	0x0
18	SYSCFG	只读	0x0
17	SPI1	只读	0x0
16	SPI0	只读	0x0
15	RTC	只读	0x0
14	PWM	只读	0x0
13	PLL_USB	只读	0x0

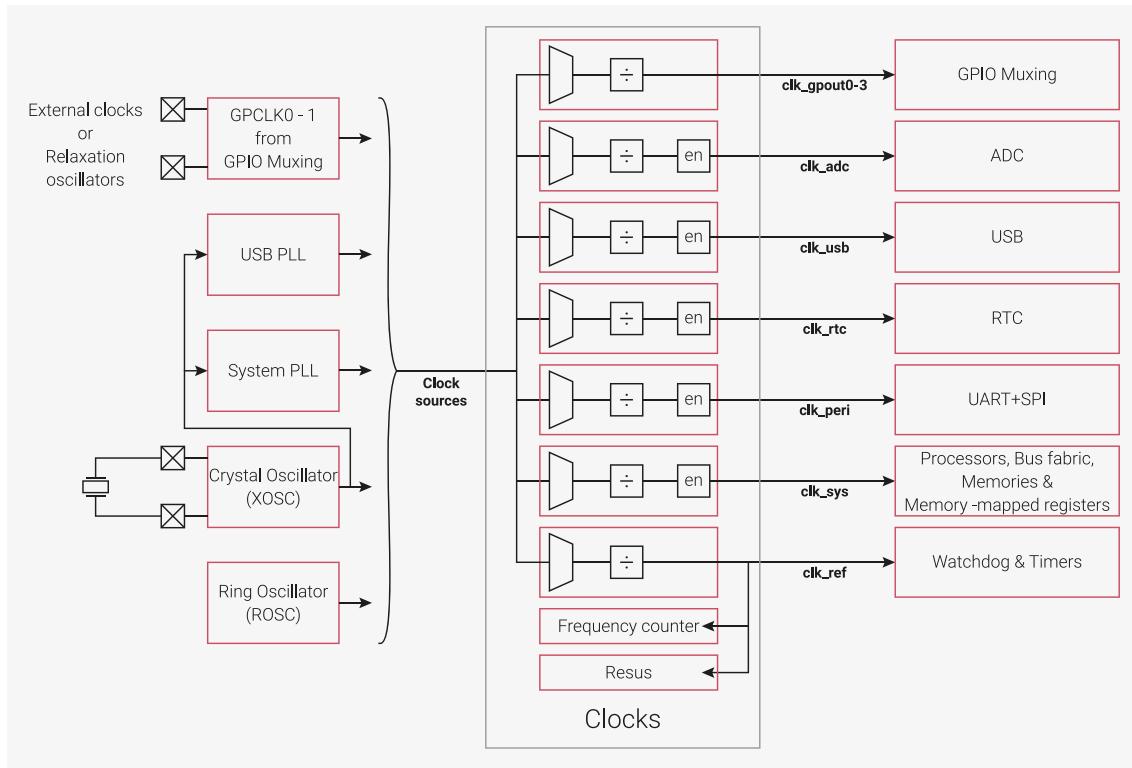
位	描述	类型	复位值
12	PLL_SYS	只读	0x0
11	PIO1	只读	0x0
10	PIO0	只读	0x0
9	PADS_QSPI	只读	0x0
8	PADS_BANK0	只读	0x0
7	JTAG	只读	0x0
6	IO_QSPI	只读	0x0
5	IO_BANK0	只读	0x0
4	I2C1	只读	0x0
3	I2C0	只读	0x0
2	DMA	只读	0x0
1	BUSCTRL	只读	0x0
0	ADC	只读	0x0

2.15. 时钟

2.15.1. 概述

时钟模块为片上及外部组件提供独立时钟。其接受多种时钟源输入，允许用户在性能、成本、板载面积及功耗间进行权衡。该模块利用多个时钟发生器，从这些时钟源产生所需时钟。该架构赋予用户灵活性，可独立启停时钟，并在保持部分时钟频率最优化的同时调整其他时钟频率。

图 28. 时钟
概述



对于不要求精确时序的极低成本或低功耗应用，芯片可由内部环形振荡器（ROSC）驱动。或者用户可提供外部时钟，亦可利用GPIO及适当的外部无源元件构建简单的回馈振荡器。在时序要求更为严格的情况下，晶体振荡器（XOSC）可为两个片内锁相环（PLL）提供精准参考，以实现快速且频率精确的时钟。

时钟发生器从时钟源中选择时钟信号，并可对选定时钟进行分频，随后通过使能逻辑输出，该逻辑在休眠模式（参见第2.11.2节）下提供自动时钟禁用功能。

片内频率计数器便于调试时钟设置，同时可测量外部时钟频率。片内复位组件在系统时钟意外停止时，能够从已知的正常时钟重新启动系统时钟。此功能使软件调试器能够访问寄存器并进行问题调试。

芯片设有名为休眠模式（参见第2.11.3节）的超低功耗模式，所有片内时钟源均被停止，以节省能耗。外部时钟源不会被停止，可用于为片内实时时钟（RTC）提供时钟，RTC能触发报警以唤醒芯片脱离休眠模式。GPIO中断亦可配置为响应外部事件，从休眠模式中唤醒芯片。

最多可将4个产生的时钟以最高50MHz的频率输出至GPIO。该功能允许用户向外部设备提供时钟，从而在对功率、空间和成本敏感的应用中减少组件数量。

2.15.2. 时钟源

RP2040可由多种时钟源提供时钟。此灵活性使用户能够针对性能、成本、电路板面积及功耗优化时钟配置。时钟源包括片上环形振荡器（第2.17节）、晶体振荡器（第2.16节）、来自GPIO的外部时钟（第2.15.6.4节）以及锁相环（第2.18节）。

各时钟发生器支持的时钟源列表不同，详见 [CTRL](#) 寄存器中的枚举值。

参见 [CLK_SYS_CTRL](#) 作为示例。

2.15.2.1. 环形振荡器

片上环振荡器（第2.17节）无须外部元件。该振荡器在上电时自动启动，用于芯片初始启动阶段的时钟信号。启动频率通常为6MHz，但会随PVT（工艺、供电电压和温度）而变化。频率大致在4至8MHz范围内，且保证处于1.8至12MHz范围内。

对于频率精度要求不高的低成本应用，芯片可继续由ROSC供时。如需更高性能，可按照第2.17节所述通过编程寄存器提升频率。频率会随PVT（工艺、供电电压和温度）波动，用户须谨慎避免超出时钟发生器部分规定的最大频率。若用户希望以接近最大频率继续由ROSC运行，可通过多种方法减缓此类波动（详见第2.15.2.1.1节）。用户也可选择使用外部时钟或XOSC提供稳定的参考时钟，并通过PLL生成更高频率。这将需要额外的外部元件，导致电路板面积增加及功耗上升。

若使用外部时钟或XOSC，则可停止ROSC以节约功耗。但必须先将参考时钟发生器和系统时钟发生器切换至备用时钟源。

ROSC不受SLEEP模式影响。如有必要，可在进入SLEEP模式之前降低频率以节约功耗。进入休眠模式时，ROSC会自动停止；退出休眠模式后，ROSC将以相同配置重新启动。若ROSC驱动的时钟频率接近最大值，建议在进入SLEEP或休眠模式之前降低频率，以适应环境变化导致的频率漂移。

若用户需要将ROSC时钟输出至外部，可使用clk_gpclk0-3发生器之一，将时钟输出到GPIO引脚。

以下章节阐述了缓解ROSC频率PVT变化的技术方案。这些方案同样为教学PVT效应及编写控制实时功能软件提供了有价值的设计挑战。

i 注意

ROSC频率随PVT变化，用户可将其输出接入频率计，在已知另外两个变量条件下，测量三者中的任一变量。

2.15.2.1.1. 缓解因工艺导致的ROSC频率波动

工艺参数变化主要源于两个方面。其一，芯片出厂时工艺参数存在差异，导致芯片间ROSC频率存在变化。其二，芯片随使用时间增长，工艺参数亦会略有变化，但该变化仅在数千小时运行后才可观察到。为缓解工艺变化，用户可针对单颗芯片进行特性测定，并据此编程调整ROSC频率。该方案适用于少量芯片，但不适合大规模批量生产。在此类应用中，用户应考虑采用下述自动缓解技术。

2.15.2.1.2. 缓解电压引起的ROSC频率变化

电源电压的变化有两个原因。首先，电源本身可能波动；其次，随着芯片活动的变化，芯片内部的IR压降也会变化。若应用具有最低性能目标，用户需针对该应用进行校准，并调整ROSC频率，以确保其始终高于最低要求。

2.15.2.1.3. 缓解温度引起的ROSC频率变化

温度变化有两个原因。首先，环境温度可能变化；其次，芯片温度会因自发热随着芯片活动而变化。此问题可通过温度控制环境以及被动或主动冷却措施来加以缓解。

控制环境和冷却。或者，用户可利用片上温度传感器监测温度，并调整ROSC频率，使其保持在所需范围内。

2.15.2.1.4 由于PVT引起的ROSC频率变化的自动缓解

自动ROSC频率控制技术避免了单独校准芯片的需求，但需要周期性地访问时钟参考或时间参考。若时钟参考可用，可周期性地测量ROSC频率并据此调整。该参考信号可为片上XOSC，且可为此目的周期性开启。此方法在功耗极低的应用中尤其有用，当连续运行XOSC代价过高，且使用PLL实现高频同样代价昂贵时。若时间参考可用，用户可用ROSC为片上RTC提供时钟，周期性将其与时间参考比较，并据此调整ROSC频率。应用此技术时，ROSC频率会因VT变化而漂移，用户必须确保这种变化不会使ROSC频率超出可接受范围。

2.15.2.1.5 使用ROSC进行自动超频

任何数字器件的数据手册中所列最大频率均基于最恶劣的PVT条件。大多数芯片在正常环境下的实际运行速度通常显著高于此最大值，因此可进行超频。若RP2040由ROSC驱动，则ROSC及数字组件均受PVT影响，随着ROSC频率提升，处理器同样得以加速。这意味着用户可通过ROSC超频，并依赖ROSC频率随PVT变化进行跟踪。ROSC频率与处理器性能的跟踪尚不完善，目前缺乏足够数据以明确该运行模式下的安全ROSC设置，因而需进行一定实验。

该运行模式虽可最大化处理器性能，但任务完成时间将出现波动，可能不适用于部分应用场景。此外，若用户需使用对频率敏感的接口（如USB或UART），则必须采用XOSC及PLL为这些组件提供精确时钟。

2.15.2.2. 晶体振荡器

晶体振荡器（第2.16节）提供精确且稳定的时钟参考，应在需要准确计时且无合适外部时钟时使用。频率由外部晶体决定，振荡器支持1MHz至15MHz范围内的频率。片上相位锁定环（PLL）可用于合成更高频率（如有需要）。RP2040参考设计（参见《使用RP2040的硬件设计》中的最小设计示例）使用12MHz晶体。利用XOSC和PLL，片上组件能够以最高频率运行。设计中预留了足够裕量，能够容忍XOSC频率高达1000ppm的偏差。

XOSC在上电时处于非激活状态，如需使用必须通过软件启用。XOSC启动需数毫秒，软件必须等待XOSC_STABLE标志被置位后，方可启动PLL并切换任何时钟生成器至该源。在此之前，XOSC输出可能存在极短脉冲，若被使用将导致逻辑损坏。一旦运行，参考时钟（clk_ref）和系统时钟（clk_sys）可以切换至由XOSC驱动，同时ROSC可被停止以节省功耗。

XOSC不受休眠模式影响。在进入和退出休眠模式时，XOSC会以相同配置自动停止并重新启动。

若用户需将XOSC时钟输出至外部，可通过clk_gpcclk0-3生成器之一，将其输出至GPIO引脚。时钟信号不可直接从XIN或XOUT引脚获取。

2.15.2.3. 外部时钟

若您的硬件设计包含外部时钟，亦可单独或结合XOSC或ROSC，为RP2040供时。此举不仅可节省功耗，还能使RP2040组件与外部组件同步运行，简化芯片间数据传输。外部时钟可输入至

GPIN0 和 GPIN1 GPIO 输入以及 XOSC 的 XIN 输入。如果以此方式使用 XIN 输入，则必须将 XOSC 配置为传递 XIN 信号。所有三个输入均限制为 50MHz，但片上 PLL 可根据需要利用 XIN 输入合成更高频率。如果外部时钟的频率精度低于 1000ppm，则生成的时钟不应以最大频率运行，以免超出设计容限。

外部时钟开始运行后，参考时钟（clk_ref）和系统时钟（clk_sys）可以切换至外部时钟驱动，同时停止 ROSC 以节省功耗。

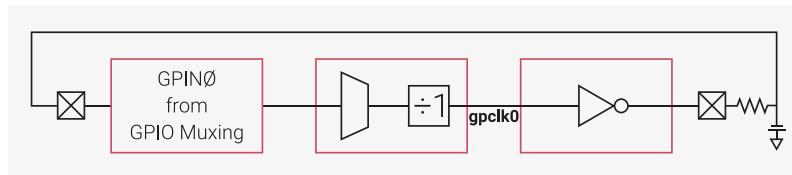
外部时钟源不受 SLEEP 模式或休眠模式影响。

2.15.2.4. 松弛振荡器

若用户希望使用外部时钟替代或补充其他时钟源，但无合适时钟可用，则可使用外部无源元件构建一至两个松弛振荡器。

简单地将时钟源（GPIN0 或 GPIN1）发送至 gpclk0-3 生成器之一，并通过 GPIO 反相器进行反相 OUTOVER，再通过 RC 电路连接回时钟源输入。

图29。简单
松弛振荡器
示例



由松弛振荡器产生的时钟频率取决于芯片内部的延迟及 GPIO 输出的驱动电流，二者均随 PVT 变化。此外，还取决于外部元件的质量与精度。虽可通过使用如陶瓷谐振器等更复杂的外部元件来提升频率精度，但这将增加成本和复杂度，且始终无法媲美 XOSC。因此，此处不予讨论。鉴于这些振荡器无法达到 1000ppm 的精度，故不可用于驱动内部时钟达到最大频率。

松弛振荡器不受 SLEEP 模式及休眠模式影响。

2.15.2.5. PLLs

PLL（参见第2.18节）用于在从XOSC（或驱动至XIN引脚的外部时钟源）运行时提供高速时钟。在功能完备的应用中，USB PLL为ADC和USB提供固定的48MHz时钟，而clk_rtc和clk_ref由XOSC或外部时钟源驱动。这允许用户通过系统PLL驱动clk_sys，并根据需求调节频率以节能，而无需更改其他时钟的设置。clk_peri可以由固定频率的USB PLL或可变频率的系统PLL驱动。如果clk_sys频率不超过48MHz，则可仅使用一个PLL，并通过clk_sys时钟生成器中的分频器根据需求调整clk_sys频率。

PLL启动后，在PLL锁定并由STATUS寄存器的LOCK位指示之前，其输出不可用。

此后，在更改参考时钟分频器、输出分频器或旁路模式时，PLL输出不可用。在反馈除数改变期间，输出可被使用，但在反馈除数发生较大变化时，输出频率可能会出现超过或低于设定值的情况。详见第2.18节。

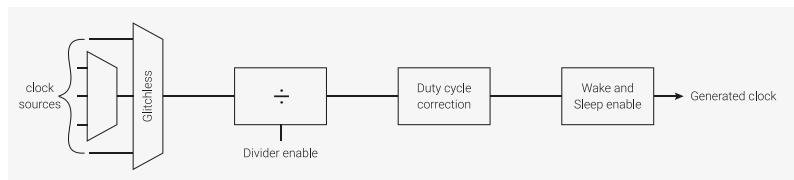
若PLL参考时钟精度达到1000ppm，则PLL可用于以最大频率驱动时钟，因为生成时钟的频率将处于设计允许的范围内。

PLL不受休眠模式影响。若用户欲在休眠模式下节能，必须将所有时钟发生器切换离开PLL，且在进入休眠模式前通过软件停止它们。进入及退出休眠模式时，PLL不会自动停止或重启。若在进入休眠模式时PLL仍运行，PLL将被破坏，产生失控时钟，导致不必要的功耗。这是因其参考时钟XOSC会被停止。因此，进入休眠模式前，务必把所有时钟发生器切换离开PLL，并通过软件停止PLL。

2.15.3. 时钟发生器

时钟发生器采用标准设计，集成时钟源多路复用、分频、占空比校正及休眠模式启用功能。为节省芯片面积和功耗，单个时钟发生器并不支持所有功能。

图30。通用时钟发生器



2.15.3.1. 实例

RP2040 配备多个时钟发生器，列举如下。

表205. RP2040
时钟发生器

时钟	描述	额定频率
<code>clk_gpout0</code>	时钟信号输出至 GPIO，可用于驱动外部设备，或使用逻辑分析仪及示波器调试片上时钟。	不适用
<code>clk_gpout1</code>		
<code>clk_gpout2</code>		
<code>clk_gpout3</code>		
<code>clk_ref</code>	参考时钟，除休眠模式外始终运行。上电时由环形振荡器（ROSC）驱动，可切换至晶体振荡器（XOSC）以提升精度。	6 - 12MHz
<code>clk_sys</code>	系统时钟，除休眠模式外始终运行。上电时由 <code>clk_ref</code> 驱动，通常切换至 PLL。	125MHz
<code>clk_peri</code>	外设时钟。通常由 <code>clk_sys</code> 运行，但若软件更改 <code>clk_sys</code> ，则允许外围设备以恒定速度运行。	12 - 125MHz
<code>clk_usb</code>	USB参考时钟，必须为48MHz。	48MHz
<code>clk_adc</code>	ADC参考时钟，必须为48MHz。	48MHz
<code>clk_rtc</code>	RTC参考时钟。RTC通过分频该时钟以产生1秒参考信号。	46875Hz

① 注意

`clk_sys`（及 `clk_peri`）在所有工艺、电压及温度变化条件下的最高频率为133MHz。
通过提升核心电源（DVDD）并将VREG VSEL设置为1.15V，可实现200MHz频率。详见第2.10.6节。

有关每个时钟发生器之完整时钟源列表，请参阅相应的 `CTRL` 寄存器。例如，`CLK_SYS_CTRL`。

2.15.3.2. 多路复用器

所有时钟发生器均设有称为辅助（aux）多路复用器的复用器，该多路复用器采用传统设计，切换选择控制时输出会产生毛刺。两个时钟发生器（`clk_sys`和`clk_ref`）设有一个额外的多路复用器，称为无毛刺多路复用器。该无毛刺多路复用器能够在切换时钟源时，避免在输出端产生毛刺。

必须全力避免时钟毛刺，因为其可能损坏基于该时钟运行的逻辑电路。这意味着，任何仅配备辅助多路复用器的时钟发生器，在切换时钟源时必须被禁用。若时钟发生器配备无毛刺多路复用器（`clk_sys`和`clk_ref`），则在更改辅助多路复用器的信号源时，应先通过无毛刺多路复用器切换离开辅助多路复用器。时钟发生器需耗费源时钟两个周期以停止输出，且需两个新源时钟周期以重新启动输出。用户必须等待发生器完全停止后方可更改辅助多路复用器，故需准确掌握源时钟频率。

无毛刺多路复用器仅适用于始终开启的时钟信号。在RP2040芯片中，始终开启的时钟包括参考时钟（`clk_ref`）及系统时钟（`clk_sys`）。此类时钟必须持续运行，除非芯片处于休眠模式。无误差复用器具有状态输出（SELECTED），用于指示所选源，且可通过软件读取以确认时钟源切换已完成。

推荐的控制序列如下：

切换无误差复用器：

- 将无误差复用器切换至备用源
- 轮询SELECTED寄存器，直至切换完成

当发生器具备无误差复用器时，切换辅助复用器：

- 将无误差复用器切换至非辅助复用器的源
- 轮询SELECTED寄存器，直至切换完成
- 更改辅助复用器选择控制
- 将无误差复用器切換回辅助复用器
- 如有需要，轮询SELECTED寄存器，直至切换完成

当发生器不具备无误差复用器时，切换辅助复用器：

- 禁用时钟分频器
- 等待生成的时钟停止（时钟源的两个周期）
- 更改辅助复用器选择控制
- 启用时钟分频器
- 如有需要，等待时钟发生器重启（时钟源的两个周期）

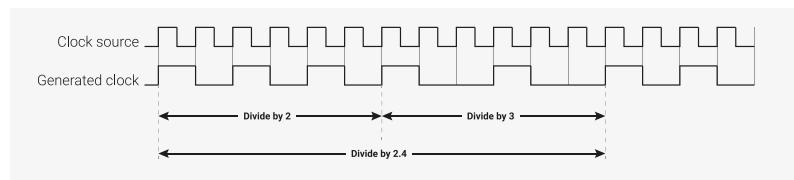
详见第2.15.6.1节代码示例。

2.15.3.3. 分频器

一个功能齐全的分频器可将频率除以1或介于2.0至 $2^{24}-0.01$ 之间的分数。分数除法通过在两个整数除数之间切换实现，因此输出时钟可能会产生抖动，某些应用可能不适用。

例如，当除数为2.4时，分频器将连续3个周期除以2，随后连续2个周期除以3。对于具有较大整数成分的除数，抖动显著减小且影响较小。

图31。分数除法的示例。



所有分频器均支持动态变更除数，输出时钟可平滑切换至新除数。

时钟发生器在改变时钟除数时无需停止。此功能通过将除数变更同步至时钟周期末实现。同样，启用信号与时钟周期结束同步，因此在时钟生成器启用或禁用时不会产生毛刺。用于始终开启的时钟生成器永久启用，因此不存在启用控制。

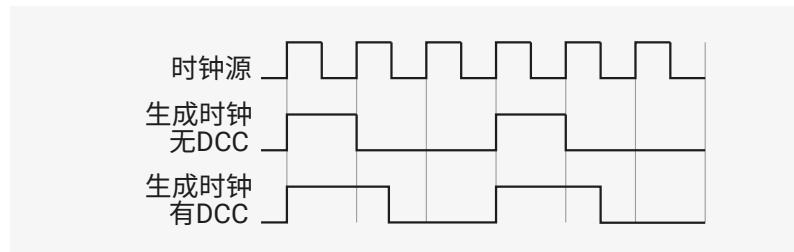
如果时钟生成器锁死，且当前时钟周期永远无法完成，可通过 KILL 控制强制停止。此操作可能导致输出毛刺，进而破坏由该时钟驱动的逻辑。因此，建议在执行此操作前先复位目标逻辑。值得一提的是，该时钟生成器设计已应用于众多芯片，且从未出现锁死情况。KILL 控制方法不够优雅且无必要，切勿用作启用控制的替代方案。始终开启的时钟生成器永久激活，因此不设 KILL 控制。

2.15.3.4. 占空比校正

分频器于输入时钟上升沿工作，故除数为奇数时无法生成均匀占空比的时钟信号。

除以3时占空比为33.3%，除以5时为40%，依此类推。如启用，占空比校正逻辑将把输出时钟的下降沿调整至输入时钟的下降沿，从而恢复50%的占空比。占空比校正功能可在时钟运行时启用或禁用。除以偶数时，该功能无效。

图32。duty_cycle_correction 的示例。



2.15.3.5 时钟使能

每个时钟输出连接至多个目标，除少数例外，每个目标具备两个使能信号。`WAKE_EN`寄存器用于系统唤醒状态下的时钟使能，`SLEEP_EN`寄存器用于系统睡眠状态下的时钟使能。设置这些使能旨在降低未使用组件时钟分布网络的功耗。值得注意的是，未被时钟驱动的组件将保留其配置，因此能够快速重启。

注意

寄存器 `WAKE_EN` 和 `SLEEP_EN` 复位为 `0x1`，意味着默认情况下所有时钟均处于启用状态。程序员仅在设计低功耗方案时需要使用此功能。

2.15.3.5.1 时钟启用例外

处理器内核无时钟启用选项，因其始终需要时钟以管理自身节能特性。

`clk_sys_busfabric`不可在唤醒模式下禁用，否则将阻止内核访问包括时钟启用控制在内的任何芯片寄存器。

`clk_sys_clocks`无唤醒模式启用选项，禁用该时钟将阻止内核访问时钟控制寄存器。

gpclks无时钟启用功能。

2.15.3.5.2 系统睡眠模式

当两个内核均处于睡眠状态且DMA无未完成事务时，系统将自动进入睡眠模式。在系统睡眠模式下，前述时钟使能从`WAKE_EN`寄存器切换至`SLEEP_EN`寄存器。其目的是在芯片处于非活动状态时减少时钟分配网络的功耗。如果用户未配置`WAKE_EN`和`SLEEP_EN`寄存器，系统睡眠将不会产生任何效果。

在内核进入睡眠之前，如未采取其他降低功耗的措施，单独使用系统睡眠意义有限。需考虑的事项包括：

- 停止未使用的时钟源，例如PLL和晶体振荡器
- 通过增加时钟分频器降低生成时钟的频率
- 停止外部时钟

为实现芯片非活动状态下的最大功耗节省，用户应考虑使用DORMANT模式（详见第2.11.3节），该模式下时钟源为晶体振荡器和/或环形振荡器，并停止这些时钟源。

2.15.4. 频率计数器

频率计数器通过在测试间隔内计数时钟的边沿数来测量内外部时钟频率。测试间隔由`clk_ref`的周期数定义，`clk_ref`必须由XOSC或已知频率的稳定外部源驱动。

用户可以通过`FC0_INTERVAL`寄存器在准确度与测试时间之间进行选择。表206展示了该折衷关系。

表206。频率计测试间隔与准确度

间隔寄存器	测试间隔	准确度
0	1μ秒	2048kHz
1	2μ秒	1024kHz
2	4μ秒	512kHz
3	8μ秒	256kHz
4	16μ秒	128kHz
5	32μ秒	64kHz
6	64μ秒	32kHz
7	125μ秒	16kHz
8	250μ秒	8kHz
9	500μ秒	4kHz
10	1ms	2kHz
11	2ms	1kHz
12	4ms	500Hz
13	8ms	250Hz

间隔寄存器	测试间隔	准确度
14	16ms	125Hz
15	32ms	62.5Hz

2.15.5. Resus

可能出现软件无意中停止 `clk_sys` 的情况。这通常会导致核心及片上调试器不可恢复的死锁，用户无法追踪问题。为防止此类情况，系统配备了自动复苏电路，如在用户设定的时间间隔内未检测到边沿，将自动将 `clk_sys` 切换至已知的良好时钟源。该良好时钟源为 `clk_ref`，可由 XOSC、ROSC 或外部源驱动。

复苏模块在由 `clk_ref` 控制的超时区间内计数 `clk_sys` 的边沿，若未检测到任何边沿，强制 `clk_sys` 由 `clk_ref` 驱动。此时间间隔可通过 `CLK_SYS_RESUS_CTRL` 寄存器进行编程设置。

● 警告

若 `clk_ref` 亦停止，复苏电路无法恢复芯片功能。

为启用 resus，程序员必须设置超时间隔，然后在 `CLK_SYS_RESUS_CTRL` 中设置 `ENABLE` 位。

为检测 resus 事件，必须通过在 INTE 中设置中断使能位启用 `CLK_SYS_RESUS` 中断。处理器还必须启用 `CLOCKS_DEFAULT_IRQ`（详见第 2.3.2 节）。

Resus 旨在作为调试辅助工具，目的是让用户追踪触发 resus 的软件错误，进而修正该错误并重启系统。通过重新配置 `clk_sys` 后，写入 `CLK_SYS_RESUS_CTRL` 中的 `CLEAR` 位即可清除 resus 并继续运行。但应注意，resus 可能由 `clk_sys` 运行速度低于预期引发，进而导致 resus 触发时出现 `clk_sys` 故障。

该故障可能会损坏芯片。此类事件虽属罕见，但在调试环境中可被接受。然而，该操作在正常运行中不可接受，因此建议仅将 resus 用于调试。

● 警告

Resus 为调试辅助工具，不应用于正常运行中的时钟切换。

2.15.6. 程序员模型

2.15.6.1. 配置时钟发生器

SDK 定义了一个时钟枚举类型：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h 第 30 行至 42 行

```

30 typedef enum clock_num_rp2040 {
31     clk_gpout0 = 0, ///< 选择 CLK_GPOUT0 作为时钟源
32     clk_gpout1 = 1, ///< 选择 CLK_GPOUT1 作为时钟源
33     clk_gpout2 = 2, ///< 选择 CLK_GPOUT2 作为时钟源
34     clk_gpout3 = 3, ///< 选择 CLK_GPOUT3 作为时钟源
35     clk_ref = 4, ///< 选择 CLK_REF 作为时钟源
36     clk_sys = 5, ///< 选择 CLK_SYS 作为时钟源
37     clk_peri = 6, ///< 选择 CLK_PERI 作为时钟源
38     clk_usb = 7, ///< 选择 CLK_USB 作为时钟源
39     clk_adc = 8, ///< 选择 CLK_ADC 作为时钟源
40     clk_rtc = 9, ///< 选择 CLK_RTC 作为时钟源
41     CLK_COUNT

```

```
42 } clock_num_t;
```

此外，还有一个结构体用于描述时钟发生器寄存器：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h 第 100 - 121 行

```
100 typedef struct {
101     _REG_(CLOCKS_CLK_GPOUT0_CTRL_OFFSET) // CLOCKS_CLK_GPOUT0_CTRL
102     // 时钟控制，除辅助源外可动态更改
103     // 0x00100000 [20]      NUDGE        (0) 此信号的一个边缘使相位发生
104     // 偏移...
105     // 0x00030000 [17:16] PHASE        (0x0) 使能信号延迟最多3个周期
106     // 的...
107     // 0x00001000 [12]      DC50         (0) 启用奇数除数的占空比校正
108     // 0x00000800 [11]      ENABLE        (0) 干净启动和停止时钟生成器
109     // 0x00000400 [10]      KILL         (0) 异步终止时钟生成器
110     // 0x000001e0 [8:5]     AUXSRC       (0x0) 选择辅助时钟源，切换时将出现抖动
111     // when switching
112     io_rw_32 ctrl;
113
114     _REG_(CLOCKS_CLK_GPOUT0_DIV_OFFSET) // CLOCKS_CLK_GPOUT0_DIV
115     // 时钟除数，可动态调整
116     // 0xfffffff00 [31:8]    INT          (0x000001) 除数的整数部分，0 表示
117     // 除以 2^16
118     // 0x000000ff [7:0]     FRAC         (0x00) 除数的小数部分
119     io_rw_32 div;
120
121     _REG_(CLOCKS_CLK_GPOUT0_SELECTED_OFFSET) // CLOCKS_CLK_GPOUT0_SELECTED
122     // 指示当前由无毛刺多路复用器选择的 SRC (单热编码)
123     // 0xffffffff [31:0] CLK_GPOUT0_SELECTED (0x00000001) 此片段不包含无毛刺多路复用器 (仅..
124     . 120 io_ro_32 selected;
125
126 } clock_hw_t;
```

配置时钟时，需了解以下信息：

- 时钟源频率
- 时钟源多路复用器/辅助多路复用器的位置
- 期望输出频率

该 SDK 提供了 `clock_configure` 函数用于配置时钟：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第40至133行

```
40 static void clock_configure_internal(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
41                                     uint32_t actual_freq, uint32_t div) {
42     clock_hw_t *clock_hw = &clocks_hw->clk[clock];
43
44     // 如果分频系数增大，先设置分频系数，再设置时钟源。否则先设置时钟源
45     // 再设置分频系数。该方式避免了例如切换
46     // 到更快时钟源且增加分频系数以补偿时钟源出现的瞬时超频现象。
47     if (div > clock_hw->div)
48         clock_hw->div = div;
49
50     // 若切换无毛刺片段（参考时钟或系统时钟）至辅助时钟源时，需先
51     // 退出辅助源，以避免更改辅助多路复用器时产生毛刺。
52     // 假设 (!!!) 无毛刺源0不比辅助源更快。
53     if (has_glitchless_mux(clock) && src ==
54         CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX) {
55         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_REF_CTRL_SRC_BITS);
```

```

54         while (!(clock_hw->selected & 1u))
55             tight_loop_contents();
56     }
57     // 如果没有无毛刺复用器，应干净停止时钟以避免
58     // 更改辅助复用器时出现毛刺传播。请注意，
59     // 在无毛刺时钟（如clk_sys、clk_ref）上执行此操作极其不妥。
60     else {
61         // 禁用时钟。对clk_ref和clk_sys无效，
62         // 其他时钟的ENABLE位均处于同一位置。
63         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
64         if (configured_freq[clock] > 0) {
65             // 延迟目标时钟3个周期，以保证ENABLE信号传播。
66             // 注意XOSC_COUNT在此处无效，因为XOSC未必在运行，定时器亦未必...
67             // 必须运行，定时器也是如此...
68             uint delay_cyc = configured_freq[clk_sys] / configured_freq[clock] + 1;
69             busy_wait_at_least_cycles(delay_cyc * 3);
70         }
71     }
72
73     // 先设置辅助多路复用器，如若时钟含无故障多路复用器，则随后设置该多路复用器
74     hw_write_masked(&clock_hw->ctrl,
75                     (auxsrc << CLOCKS_CLK_SYS_CTRL_AUXSRC_LSB),
76                     CLOCKS_CLK_SYS_CTRL_AUXSRC_BITS
77     );
78
79     if (has_glitchless_mux(clock)) {
80         hw_write_masked(&clock_hw->ctrl,
81                         src << CLOCKS_CLK_REF_CTRL_SRC_LSB,
82                         CLOCKS_CLK_REF_CTRL_SRC_BITS
83         );
84         while (!(clock_hw->selected & (1u << src)))
85             tight_loop_contents();
86     }
87
88     // 启用时钟。对clk_ref和clk_sys无效，
89     // 所有其他时钟的ENABLE位均位于相同位置。
90     hw_set_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
91
92     // 既然源已配置完毕，我们可以信赖用户提供的
93     // 除数为安全值。
94     clock_hw->div = div;
95     configured_freq[clock] = actual_freq;
96 }
97
98 bool clock_configure(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq,
99                      uint32_t freq) {
100    assert(src_freq >= freq);
101
102    if (freq > src_freq)
103        return false;
104
105    uint64_t div64 = (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) / freq;
106    uint32_t div, actual_freq;
107    if (div64 > 32) {
108        // 将除数设置为0，表示最大时钟分频
109        div = 0;
110        actual_freq = src_freq >> (32 - CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
111    } else {
112        div = (uint32_t) div64;
113        // 仅在RP2040上支持时钟分频为1或不小于2的设置
114        if (div < (2u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB)) {
115            div = (1u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
116        }
117        actual_freq = (uint32_t) (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) /

```

```

        div);
117    }
118
119    clock_configure_internal(clock, src, auxsrc, actual_freq, div);
120    // 存储已配置的频率
121    return true;
122 }
123
124 void clock_configure_int_divider(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
125     uint32_t src_freq, uint32_t int_divider) {
126     clock_configure_internal(clock, src, auxsrc, src_freq / int_divider, int_divider <<
127     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
128 }
129
128 void clock_configure_undivided(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t
129     src_freq) {
130     clock_configure_internal(clock, src, auxsrc, src_freq, 1u <<
131     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
132 }

```

该函数在`clocks_init`中针对每个时钟调用。以下示例展示了`clk_sys`的配置：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_runtime_init/runtime_init_clocks.c 第100至104行

```

100     // CLK SYS = PLL SYS (通常) 125MHz / 1 = 125MHz
101     clock_configure_undivided(clk_sys,
102         CLOCKSYS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
103         CLOCKSYS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS,
104         SYS_CLK_HZ);

```

时钟配置完成后，可调用`clock_get_hz`获取输出频率（单位 Hz）。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第137-139行

```

137 uint32_t clock_get_hz(clock_handle_t clock) {
138     return configured_freq[clock];
139 }

```

● 警告

假定程序员提供的源频率是正确的。若不正确，则`clock_get_hz`返回的频率将不准确。

2.15.6.2. 频率计的使用

使用频率计时，程序员必须：

- 设置参考频率：`clk_ref`
- 设置欲测量的信号源的复用器位置。详见FC0_SRC
- 等待FC0_STATUS中`DONE`状态位被置位
- 读取测量结果

SDK定义了一个`frequency_count`函数，参数为信号源，返回以kHz为单位的频率：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第147-174行

```

147 uint32_t frequency_count_khz(uint src) {
148     fc_hw_t *fc = &clocks_hw->fc0;
149
150 // 如果频率计正在运行，需等待其完成。即使源为NULL，频率计仍然运行 151 while(fc->status & CLOCKS_FC0_STATUS_RUNNING_BITS) {
152     tight_loop_contents();
153 }
154
155 // 设置参考频率
156 fc->ref_khz = clock_get_hz(clk_ref) / 1000;
157
158 // 待修正：不要选择随机间隔。应使用最佳间隔 159 fc->interval = 10;
159
160
161 // 无最小或最大值限制
162 fc->min_khz = 0;
163 fc->max_khz = 0xffffffff;
164
165 // 设置 SRC，自动开始测量
166 fc->src = src;
167
168 while(!(fc->status & CLOCKS_FC0_STATUS_DONE_BITS)) {
169     tight_loop_contents();
170 }
171
172 // 返回结果
173 return fc->result >> CLOCKS_FC0_RESULT_KHZ_LSB;
174 }
```

还提供了一个包装函数，用于将单位转换为MHz：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware_clocks.h 第377至379行

```

377 static inline float frequency_count_mhz(uint src) {
378     return ((float)(frequency_count_khz(src))) / KHZ;
379 }
```

注意

频率计数器亦可在测试模式下使用。该模式允许硬件检测频率是否位于FC0_MIN_KHZ与FC0_MAX_KHZ设定的最小与最大频率之间。当频率在指定范围内且DONE位被置位时，FC0_STATUS的PASS位将被设置。否则，将设置FAST位或SLOW位。

如果程序员尝试读取已停止的时钟，或时钟停止运行，则会设置 DIED位。若DIED、FAST 或SLOW中任一标志被置位，则FAIL标志也将被设置。

2.15.6.3. 配置GPIO输出时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第245至276行

```

245 void clock_gpio_init_int_frac16(uint gpio, uint src, uint32_t div_int, uint16_t div_frac16)
{
246     // 代码较为杂乱，但遍历查找表的代码量相当
247     // 每个gout发生器的时钟源均相同
```

```

248     // 因此仅需使用GP0的时钟源调用
249     uint gpclk = 0;
250     if      (gpio == 21) gpclk = clk_gpout0;
251     else if (gpio == 23) gpclk = clk_gpout1;
252     else if (gpio == 24) gpclk = clk_gpout2;
253     else if (gpio == 25) gpclk = clk_gpout3;
254     else {
255         invalid_params_if(HARDWARE_CLOCKS, true);
256     }
257
258     invalid_params_if(HARDWARE_CLOCKS, div_int >> REG_FIELD_WIDTH(
259         CLOCKS_CLK_GPOUT0_DIV_INT));
260     // 设置 gpclk 生成器
261     clocks_hw->clk[gpclk].ctrl = (src << CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_LSB) |
262                                     CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS;
263 #ifdef REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 16
264     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | (div_frac16 <<
265                                     CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
266 #elif REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 8
267     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | ((div_frac16
268 >>8u) << CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
269 #else
270     #error 不支持的分位数
271 #endif
272     // 设置 GPIO 引脚为 gpclock 功能
273     gpio_set_function(gpio, GPIO_FUNC_GPCK);
274 }
```

2.15.6.4. 配置 GPIO 输入时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第 313 - 343 行

```

313 bool clock_configure_gpin(clock_handle_t clock, uint gpio, uint32_t src_freq, uint32_t freq)
{
314     // 配置时钟以从 GPIO 输入运行
315     uint gpin = 0;
316     if      (gpio == 20) gpin = 0;
317     else if (gpio == 22) gpin = 1;
318     else {
319         invalid_params_if(HARDWARE_CLOCKS, true);
320     }
321
322 // 计算信号源GPIN始终为auxsrc 323 uint src = 0;

324
325     // GPIN1 == GPIN0 + 1
326     uint auxsrc = gpin0_src[clock] + gpin;
327
328     if (has_glitchless_mux(clock)) {
329         // AUX 源始终为 1
330         src = 1;
331     }
332
333     // 设置 GPIO 功能
334     gpio_set_function(gpio, GPIO_FUNC_GPCK);
335
336     // 目前已有 src、auxsrc，并完成 gpio 输入的配置
337     // 调用 clock_configure 以从 gpio 驱动时钟
338     return clock_configure(clock, src, auxsrc, src_freq, freq);
```

339 }

2.15.6.5. 启用 resus

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第 221 - 243 行

```

221 void clocks_enable_resus(resus_callback_t resus_callback) {
222     // 如果 clk_sys 被停止，则通过强制其重启
223 // 到 clk_ref 的默认时钟源。如果 clk_ref 停止运行，本方法224 // 将不起作用。
224
225     // 保存用户的 resus 回调函数
226     _resus_callback = resus_callback;
227
228     irq_set_exclusive_handler(CLOCKS_IRQ, clocks_irq_handler);
229
230     // 启用 clocks 中的 resus 中断
231     clocks_hw->inte = CLOCKS_INTE_CLK_SYS_RESUS_BITS;
232
233     // 启用 clocks 中断
234     irq_set_enabled(CLOCKS_IRQ, true);
235
236     // 2 * clk_ref 频率 / clk_sys_min_freq;
237     // 假设 clk_ref 为 3MHz，且期望 clk_sys 不低于 1MHz
238     uint timeout = 2 * 3 * 1;
239
240     // 使用最大超时时间启用复苏功能
241     clocks_hw->resus.ctrl = CLOCKS_CLK_SYS_RESUS_CTRL_ENABLE_BITS | timeout;
242
243 }

```

2.15.6.6 配置睡眠模式

当处理器核心和 DMA 均未请求时钟时，睡眠模式即处于激活状态。例如，当 DMA 不活动且 core0与 core1均在等待中断时。寄存器 SLEEP_EN用于设置睡眠模式下运行的时钟。

[示例程序hello_sleep](#) (https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c) 展示了如何使芯片进入睡眠状态，直至 RTC 触发。在此情况下，仅在 SLEEP_EN0 寄存器中启用 RTC 时钟。

① 注意

clk_sys 在睡眠模式期间始终发送至 proc0 和 proc1，因某些逻辑需时钟驱动以使处理器能够重新唤醒。

Pico 附加功能: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第159至183行

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // 我们应已调用过 sleep_run_from_dormant_source 函数
163 // 该功能仅在休眠状态下需要，尽管它在睡眠时通过 xosc 节约了功耗
164
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCKS_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;

```

```

168
169     aon_timer_enable_alarm(ts, callback, false);
170
171     stdio_flush();
172
173     // 在处理器上启用深度睡眠
174     processor_deep_sleep();
175
176     // 进入睡眠状态
177     __wfi();
178 }

```

2.15.7. 寄存器列表

时钟寄存器地址为 `0x40008000`（在 SDK 中定义为 CLOCKS_BASE）。

表207. CLOCKS
寄存器列表

偏移量	名称	说明
0x00	<code>CLK_GPOUT0_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x04	<code>CLK_GPOUT0_DIV</code>	时钟分频器，支持动态更改
0x08	<code>CLK_GPOUT0_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x0c	<code>CLK_GPOUT1_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x10	<code>CLK_GPOUT1_DIV</code>	时钟分频器，支持动态更改
0x14	<code>CLK_GPOUT1_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x18	<code>CLK_GPOUT2_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x1c	<code>CLK_GPOUT2_DIV</code>	时钟分频器，支持动态更改
0x20	<code>CLK_GPOUT2_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x24	<code>CLK_GPOUT3_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x28	<code>CLK_GPOUT3_DIV</code>	时钟分频器，支持动态更改
0x2c	<code>CLK_GPOUT3_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x30	<code>CLK_REF_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x34	<code>CLK_REF_DIV</code>	时钟分频器，支持动态更改
0x38	<code>CLK_REF_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x3c	<code>CLK_SYS_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x40	<code>CLK_SYS_DIV</code>	时钟分频器，支持动态更改
0x44	<code>CLK_SYS_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x48	<code>CLK_PERI_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x50	<code>CLK_PERI_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。

偏移量	名称	说明
0x54	CLK_USB_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x58	CLK_USB_DIV	时钟分频器，支持动态更改
0x5c	CLK_USB_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x60	CLK_ADC_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x64	CLK_ADC_DIV	时钟分频器，支持动态更改
0x68	CLK_ADC_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x6c	CLK_RTC_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x70	CLK_RTC_DIV	时钟分频器，支持动态更改
0x74	CLK_RTC_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x78	CLK_SYS_RESUS_CTRL	
0x7c	CLK_SYS_RESUS_STATUS	
0x80	FC0_REF_KHZ	参考时钟频率，单位 kHz
0x84	FC0_MIN_KHZ	最小通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0。
0x88	FC0_MAX_KHZ	最大通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0xffffffff。
0x8c	FC0_DELAY	延迟频率计数启动，以使多路复用器稳定 延迟以参考时钟周期的整数倍计量
0x90	FC0_INTERVAL	测试间隔为0.98微秒 * 2**interval，但我们称之为1微秒 * 2* *interval 默认测试间隔为250微秒
0x94	FC0_SRC	传送至频率计数器的时钟，非必要时设置为0 写入此寄存器即可启动频率计数
0x98	FC0_STATUS	频率计数器状态
0x9c	FC0_RESULT	频率测量结果，仅在status_done=1时有效
0xa0	WAKE_EN0	启用唤醒模式下的时钟
0xa4	WAKE_EN1	启用唤醒模式下的时钟
0xa8	SLEEP_EN0	启用睡眠模式下的时钟
0xac	SLEEP_EN1	启用睡眠模式下的时钟
0xb0	ENABLED0	指示时钟使能状态
0xb4	ENABLED1	指示时钟使能状态
0xb8	INTR	原始中断
0xbc	INTE	中断使能
0xc0	INTF	中断强制
0xc4	INTS	掩码及强制后的中断状态

CLOCKS: CLK_GPOUT0_CTRL 寄存器

偏移: 0x00

描述

时钟控制，支持动态更改（auxsrc 除外）

表208
CLK_GPOUT0_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_SYS 0x1 → CLKSRC_GPIO 0x2 → CLKSRC_GPIO1 0x3 → CLKSRC_PLL_USB 0x4 → ROSC_CLKSRC 0x5 → XOSC_CLKSRC 0x6 → CLK_SYS 0x7 → CLK_USB 0x8 → CLK_ADC 0x9 → CLK_RTC 0xa → CLK_REF	读写	0x0
4:0	保留。	-	-

CLOCKS: CLK_GPOUT0_DIV 寄存器

偏移量: 0x04

说明

时钟分频器，支持动态更改

表 209。
CLK_GPOUT0_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分, 0 —除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT0_SELECTED 寄存器

偏移量: 0x08

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 210。
CLK_GPOUT0_SELEC
T_ED 寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT1_CTRL 寄存器

偏移: 0x0c

说明

时钟控制，支持动态更改（auxsrc 除外）

表 211。
CLK_GPOUT1_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_SYS 0x1 → CLKSRC_GPIO 0x2 → CLKSRC_GPIO1 0x3 → CLKSRC_PLL_USB 0x4 → ROSC_CLKSRC 0x5 → XOSC_CLKSRC 0x6 → CLK_SYS 0x7 → CLK_USB	读写	0x0

位	描述	类型	复位值
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT1_DIV 寄存器

偏移: 0x10

说明

时钟分频器，支持动态更改

表 212
CLK_GPOUT1_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT1_SELECTED 寄存器

偏移: 0x14

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 213
CLK_GPOUT1_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT2_CTRL 寄存器

偏移: 0x18

说明

时钟控制，支持动态更改（auxsrc 除外）

表 214
CLK_GPOUT2_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-

位	描述	类型	复位值
8:5	AUXSRC: 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIN0		
	0x2 → CLKSRC_GPIN1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT2_DIV 寄存器

偏移：0x1c

说明

时钟分频器，支持动态更改

表 215
CLK_GPOUT2_DIV
寄存器

位	描述	类型	复位值
31:8	INT: 除数的整数部分，0 → 除以 2^{16}	读写	0x000001
7:0	FRAC: 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT2_SELECTED 寄存器

偏移：0x20

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 216
CLK_GPOUT2_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT3_CTRL 寄存器

偏移：0x24

说明

时钟控制，支持动态更改（auxsrc 除外）

表 217。
CLK_GPOUT3_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-

位	描述	类型	复位值
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLK_SRC_PLL_SYS		
	0x1 → CLK_SRC_GPIO		
	0x2 → CLK_SRC_GPIO1		
	0x3 → CLK_SRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT3_DIV寄存器

偏移: 0x28

描述

时钟分频器，支持动态更改

表218。
CLK_GPOUT3_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 – 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT3_SELECTED寄存器

偏移: 0x2c

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表219。
CLK_GPOUT3_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_REF_CTRL 寄存器

偏移：0x30

描述

时钟控制，支持动态更改（auxsrc 除外）

表220。
CLK_REF_CTRL
寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6:5	AUXSRC ：选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIO1		
4:2	保留。	-	-
1:0	SRC ：无毛刺方式选择时钟源，支持动态切换	读写	-
	枚举值：		
	0x0 → ROOSC_CLKSRC_PH		
	0x1 → CLKSRC_CLK_REF_AUX		
	0x2 → XOSC_CLKSRC		

CLOCKS: CLK_REF_DIV 寄存器

偏移：0x34

描述

时钟分频器，支持动态更改

表 221。
CLK_REF_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT ：除数的整数部分，0 – 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_REF_SELECTED 寄存器

偏移：0x38

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 222。
CLK_REF_SELECTED
寄存器

位	描述	类型	复位值
31:0	无毛刺多路复用器的切换不是瞬时完成的（以避免毛刺），因此软件应轮询该寄存器以等待切换完成。该寄存器包含 CTRL_SRC 字段中枚举的每个时钟源的一个解码位。任一时刻最多只有一个位被置位，表示该时钟当前在无毛刺多路复用器的输出端存在。在切换过程中，该寄存器可能短暂显示全零。	只读	0x00000001

CLOCKS: CLK_SYS_CTRL 寄存器

偏移: 0x3c

描述

时钟控制，支持动态更改（auxsrc 除外）

表 223。
CLK_SYS_CTRL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_PLL_USB		
	0x2 → ROSC_CLKSRC		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:1	保留。	-	-
0	SRC : 无毛刺方式选择时钟源，支持动态切换	读写	0x0
	枚举值：		
	0x0 → CLK_REF		
	0x1 → CLKSRC_CLK_SYS_AUX		

CLOCKS: CLK_SYS_DIV 寄存器

偏移: 0x40

描述

时钟分频器，支持动态更改

表 224
CLK_SYS_DIV 寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 – 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_SYS_SELECTED 寄存器

偏移: 0x44

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 225
CLK_SYS_SELECTED
寄存器

位	描述	类型	复位值
31:0	无毛刺多路复用器的切换不是瞬时完成的（以避免毛刺），因此软件应轮询该寄存器以等待切换完成。该寄存器包含 CTRL_SRC 字段中枚举的每个时钟源的一个解码位。任一时刻最多只有一个位被置位，表示该时钟当前在无毛刺多路复用器的输出端存在。在切换过程中，该寄存器可能短暂显示全零。	只读	0x00000001

CLOCKS: CLK_PERI_CTRL 寄存器

偏移: 0x48

描述

时钟控制，支持动态更改（auxsrc 除外）

表 226
CLK_PERI_CTRL
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLK_SYS		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → CLKSRC_PLL_USB		
	0x3 → ROSC_CLKSRC_PH		
	0x4 → XOSC_CLKSRC		
	0x5 → CLKSRC_GPIN0		
	0x6 → CLKSRC_GPIN1		
4:0	保留。	-	-

时钟：CLK_PERI_SELECTED 寄存器

偏移：0x50

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 227.
CLK_PERI_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

时钟：CLK_USB_CTRL 寄存器

偏移：0x54

说明

时钟控制，支持动态更改（auxsrc 除外）

表 228.
CLK_USB_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	保留。	-	-

时钟：CLK_USB_DIV 寄存器

偏移：0x58

说明

时钟分频器，支持动态更改

表 229.
CLK_USB_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_USB_SELECTED 寄存器

偏移：0x5c

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 230
CLK_USB_SELECTED 寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_ADC_CTRL 寄存器

偏移: 0x60

描述

时钟控制，支持动态更改（auxsrc 除外）

表 231
CLK_ADC_CTRL 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	保留。	-	-

CLOCKS: CLK_ADC_DIV 寄存器

偏移: 0x64

描述

时钟分频器，支持动态更改

表 232
CLK_ADC_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_ADC_SELECTED 寄存器

偏移: 0x68

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 233.
CLK_ADC_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_RTC_CTRL 寄存器

偏移: 0x6c

描述

时钟控制，支持动态更改（auxsrc 除外）

表 234.
CLK_RTC_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_USB 0x1 → CLKSRC_PLL_SYS 0x2 → ROSC_CLKSRC_PH 0x3 → XOSC_CLKSRC 0x4 → CLKSRC_GPIN0 0x5 → CLKSRC_GPIN1	读写	0x0
4:0	保留。	-	-

CLOCKS: CLK_RTC_DIV 寄存器

偏移: 0x70

描述

时钟分频器，支持动态更改

表 235.
CLK_RTC_DIV 寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分, 0 一除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_RTC_SELECTED 寄存器

偏移: 0x74

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 236.
CLK_RTC_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_SYS_RESUS_CTRL 寄存器

偏移: 0x78

表 237.
CLK_SYS_RESUS_CTRL
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	CLEAR : 在触发故障已纠正后清除复位	读写	0x0
15:13	保留。	-	-
12	FRCE : 强制复位, 仅限测试用途	读写	0x0
11:9	保留。	-	-
8	ENABLE : 启用复位	读写	0x0
7:0	TIMEOUT : 以clk_ref周期数表示, 且必须 $\geq 2 \times \text{clk_ref_freq} / \text{min_clk_tst_freq}$	读写	0xff

CLOCKS: CLK_SYS_RESUS_STATUS 寄存器

偏移: 0x7c

表 238.
CLK_SYS_RESUS_STATUS
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RESUSSED : 时钟已恢复, 纠正错误后发送 <i>ctrl_clear=1</i>	只读	0x0

CLOCKS: FC0_REF_KHZ 寄存器

偏移: 0x80

表239
FC0_REF_KHZ寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:0	参考时钟频率，单位 kHz	读写	0x00000

CLOCKS: FC0_MIN_KHZ寄存器

偏移: 0x84

表240
FC0_MIN_KHZ
Z寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24:0	最小通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0。	读写	0x0000000

CLOCKS: FC0_MAX_KHZ寄存器

偏移: 0x88

表241。
FC0_MAX_KHZ
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24:0	最大通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0x1ffff fff。	读写	0x1fffffff

时钟：FC0_DELAY寄存器

偏移: 0x8c

表242. FC0_DELAY
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2:0	延迟频率计数启动，以使多路复用器稳定 延迟以参考时钟周期的整数倍计量	读写	0x1

时钟：FC0_INTERVAL寄存器

偏移: 0x90

表243。
FC0_INTERVAL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	测试间隔为0.98微秒 * 2**interval，但我们称之为1微秒 * 2**interval 默认测试间隔为250微秒	读写	0x8

时钟：FC0_SRC寄存器

偏移: 0x94

表244. FC0_SRC
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	传送至频率计数器的时钟，非必要时设置为0 写入此寄存器即可启动频率计数	读写	0x00
	枚举值：		
	0x00 → NULL		

位	描述	类型	复位值
0x01	→ PLL_SYS_CLKSRC_PRIMARY		
0x02	→ PLL_USB_CLKSRC_PRIMARY		
0x03	→ ROSC_CLKSRC		
0x04	→ ROSC_CLKSRC_PH		
0x05	→ XOSC_CLKSRC		
0x06	→ CLKSRC_GPIN0		
0x07	→ CLKSRC_GPIN1		
0x08	→ CLK_REF		
0x09	→ CLK_SYS		
0x0a	→ CLK_PERI		
0x0b	→ CLK_USB		
0x0c	→ CLK_ADC		
0x0d	→ CLK_RTC		

CLOCKS: FC0_STATUS 寄存器

偏移: 0x98

说明

频率计数器状态

表 245。
FC0_STATUS 寄存器

位	描述	类型	复位值
31:29	保留。	-	-
28	DIED : 测试时钟在测试过程中停止	只读	0x0
27:25	保留。	-	-
24	FAST : 测试时钟快于预期, 仅当 status_done=1 时有效	只读	0x0
23:21	保留。	-	-
20	SLOW : 测试时钟慢于预期, 仅当 status_done=1 时有效	只读	0x0
19:17	保留。	-	-
16	FAIL : 测试失败	只读	0x0
15:13	保留。	-	-
12	等待中: 等待测试时钟启动	只读	0x0
11:9	保留。	-	-
8	运行中: 测试运行中	只读	0x0
7:5	保留。	-	-
4	完成: 测试完成	只读	0x0
3:1	保留。	-	-
0	通过: 测试通过	只读	0x0

时钟: FC0_RESULT 寄存器

偏移: 0x9c

描述

频率测量结果，仅当 status_done=1 时有效

表 246.
FC0_RESULT 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:5	KHZ	只读	0x00000000
4:0	FRAC	只读	0x00

时钟： WAKE_EN0 寄存器

偏移: 0xa0

描述

启用唤醒模式下的时钟

表247. WAKE_EN0寄
存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	读写	0x1
30	CLK_SYS_SRAM2	读写	0x1
29	CLK_SYS_SRAM1	读写	0x1
28	CLK_SYS_SRAM0	读写	0x1
27	CLK_SYS_SPI1	读写	0x1
26	CLK_PERI_SPI1	读写	0x1
25	CLK_SYS_SPI0	读写	0x1
24	CLK_PERI_SPI0	读写	0x1
23	CLK_SYS_SIO	读写	0x1
22	CLK_SYS_RTC	读写	0x1
21	CLK_RTC_RTC	读写	0x1
20	CLK_SYS_ROSC	读写	0x1
19	CLK_SYS_ROM	读写	0x1
18	CLK_SYS_RESETS	读写	0x1
17	CLK_SYS_PWM	读写	0x1
16	CLK_SYS_PSM	读写	0x1
15	CLK_SYS_PLL_USB	读写	0x1
14	CLK_SYS_PLL_SYS	读写	0x1
13	CLK_SYS_PIO1	读写	0x1
12	CLK_SYS_PIO0	读写	0x1
11	CLK_SYS_PADS	读写	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	读写	0x1
9	CLK_SYS_JTAG	读写	0x1
8	CLK_SYS_IO	读写	0x1

位	描述	类型	复位值
7	CLK_SYS_I2C1	读写	0x1
6	CLK_SYS_I2C0	读写	0x1
5	CLK_SYS_DMA	读写	0x1
4	CLK_SYS_BUSFABRIC	读写	0x1
3	CLK_SYS_BUSCTRL	读写	0x1
2	CLK_SYS_ADC	读写	0x1
1	CLK_ADC_ADC	读写	0x1
0	CLK_SYS_CLOCKS	读写	0x1

时钟：WAKE_EN1 寄存器

偏移: 0xa4

描述

启用唤醒模式下的时钟

表248. WAKE_EN1寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	读写	0x1
13	CLK_SYS_XIP	读写	0x1
12	CLK_SYS_WATCHDOG	读写	0x1
11	CLK_USB_USBCTRL	读写	0x1
10	CLK_SYS_USBCTRL	读写	0x1
9	CLK_SYS_UART1	读写	0x1
8	CLK_PERI_UART1	读写	0x1
7	CLK_SYS_UART0	读写	0x1
6	CLK_PERI_UART0	读写	0x1
5	CLK_SYS_TIMER	读写	0x1
4	CLK_SYS_TBMAN	读写	0x1
3	CLK_SYS_SYSINFO	读写	0x1
2	CLK_SYS_SYSCFG	读写	0x1
1	CLK_SYS_SRAM5	读写	0x1
0	CLK_SYS_SRAM4	读写	0x1

时钟：SLEEP_EN0 寄存器

偏移: 0xa8

描述

启用睡眠模式下的时钟

表249. SLEEP_EN0寄存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	读写	0x1

位	描述	类型	复位值
30	CLK_SYS_SRAM2	读写	0x1
29	CLK_SYS_SRAM1	读写	0x1
28	CLK_SYS_SRAM0	读写	0x1
27	CLK_SYS_SPI1	读写	0x1
26	CLK_PERI_SPI1	读写	0x1
25	CLK_SYS_SPI0	读写	0x1
24	CLK_PERI_SPI0	读写	0x1
23	CLK_SYS_SIO	读写	0x1
22	CLK_SYS_RTC	读写	0x1
21	CLK_RTC_RTC	读写	0x1
20	CLK_SYS_ROSC	读写	0x1
19	CLK_SYS_ROM	读写	0x1
18	CLK_SYS_RESETS	读写	0x1
17	CLK_SYS_PWM	读写	0x1
16	CLK_SYS_PSM	读写	0x1
15	CLK_SYS_PLL_USB	读写	0x1
14	CLK_SYS_PLL_SYS	读写	0x1
13	CLK_SYS_PIO1	读写	0x1
12	CLK_SYS_PIO0	读写	0x1
11	CLK_SYS_PADS	读写	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	读写	0x1
9	CLK_SYS_JTAG	读写	0x1
8	CLK_SYS_IO	读写	0x1
7	CLK_SYS_I2C1	读写	0x1
6	CLK_SYS_I2C0	读写	0x1
5	CLK_SYS_DMA	读写	0x1
4	CLK_SYS_BUSFABRIC	读写	0x1
3	CLK_SYS_BUSCTRL	读写	0x1
2	CLK_SYS_ADC	读写	0x1
1	CLK_ADC_ADC	读写	0x1
0	CLK_SYS_CLOCKS	读写	0x1

CLOCKS: SLEEP_EN1 寄存器

偏移: 0xac

描述

启用睡眠模式下的时钟

表250. SLEEP_EN1寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	读写	0x1
13	CLK_SYS_XIP	读写	0x1
12	CLK_SYS_WATCHDOG	读写	0x1
11	CLK_USB_USBCTRL	读写	0x1
10	CLK_SYS_USBCTRL	读写	0x1
9	CLK_SYS_UART1	读写	0x1
8	CLK_PERI_UART1	读写	0x1
7	CLK_SYS_UART0	读写	0x1
6	CLK_PERI_UART0	读写	0x1
5	CLK_SYS_TIMER	读写	0x1
4	CLK_SYS_TBMAN	读写	0x1
3	CLK_SYS_SYSINFO	读写	0x1
2	CLK_SYS_SYSCFG	读写	0x1
1	CLK_SYS_SRAM5	读写	0x1
0	CLK_SYS_SRAM4	读写	0x1

CLOCKS: ENABLEDO 寄存器

偏移: 0xb0

描述

指示时钟使能状态

表 251. ENABLEDO 寄存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	只读	0x0
30	CLK_SYS_SRAM2	只读	0x0
29	CLK_SYS_SRAM1	只读	0x0
28	CLK_SYS_SRAM0	只读	0x0
27	CLK_SYS_SPI1	只读	0x0
26	CLK_PERI_SPI1	只读	0x0
25	CLK_SYS_SPI0	只读	0x0
24	CLK_PERI_SPI0	只读	0x0
23	CLK_SYS_SIO	只读	0x0
22	CLK_SYS_RTC	只读	0x0
21	CLK_RTC_RTC	只读	0x0
20	CLK_SYS_ROSC	只读	0x0
19	CLK_SYS_ROM	只读	0x0
18	CLK_SYS_RESETS	只读	0x0

位	描述	类型	复位值
17	CLK_SYS_PWM	只读	0x0
16	CLK_SYS_PSM	只读	0x0
15	CLK_SYS_PLL_USB	只读	0x0
14	CLK_SYS_PLL_SYS	只读	0x0
13	CLK_SYS_PIO1	只读	0x0
12	CLK_SYS_PIO0	只读	0x0
11	CLK_SYS_PADS	只读	0x0
10	CLK_SYS_VREG_AND_CHIP_RESET	只读	0x0
9	CLK_SYS_JTAG	只读	0x0
8	CLK_SYS_IO	只读	0x0
7	CLK_SYS_I2C1	只读	0x0
6	CLK_SYS_I2C0	只读	0x0
5	CLK_SYS_DMA	只读	0x0
4	CLK_SYS_BUSFABRIC	只读	0x0
3	CLK_SYS_BUSCTRL	只读	0x0
2	CLK_SYS_ADC	只读	0x0
1	CLK_ADC_ADC	只读	0x0
0	CLK_SYS_CLOCKS	只读	0x0

CLOCKS: ENABLED1 寄存器

偏移: 0xb4

描述

指示时钟使能状态

表 252. ENABLED1
寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	只读	0x0
13	CLK_SYS_XIP	只读	0x0
12	CLK_SYS_WATCHDOG	只读	0x0
11	CLK_USB_USBCTRL	只读	0x0
10	CLK_SYS_USBCTRL	只读	0x0
9	CLK_SYS_UART1	只读	0x0
8	CLK_PERI_UART1	只读	0x0
7	CLK_SYS_UART0	只读	0x0
6	CLK_PERI_UART0	只读	0x0
5	CLK_SYS_TIMER	只读	0x0
4	CLK_SYS_TBMAN	只读	0x0

位	描述	类型	复位值
3	CLK_SYS_SYSINFO	只读	0x0
2	CLK_SYS_SYSCFG	只读	0x0
1	CLK_SYS_SRAM5	只读	0x0
0	CLK_SYS_SRAM4	只读	0x0

CLOCKS: INTR 寄存器

偏移: 0xb8

描述

原始中断

表 253. INTR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	只读	0x0

CLOCKS: INTF 寄存器

偏移: 0xbc

描述

中断使能

表 254. INTF
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	读写	0x0

CLOCKS: INTS 寄存器

偏移: 0xc0

描述

中断强制

表 255. INTS
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	读写	0x0

CLOCKS: INTS 寄存器

偏移: 0xc4

描述

掩码及强制后的中断状态

表 256. INT3
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	只读	0x0

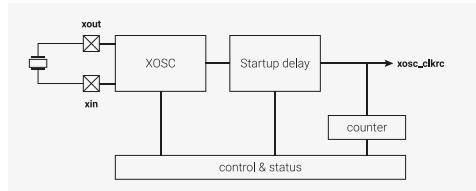
2.16. 晶体振荡器 (XOSC)

2.16.1. 概述

晶振 (XOSC) 采用外部晶体以提供精确的参考时钟。RP2040支持1MHz至15MHz晶体，RP2040参考设计（见《基于RP2040的硬件设计》中的最小设计示例）采用12MHz晶体。参考时钟分配给PLL，用于倍频XOSC频率，提供精确的高速时钟。例如，PLL可生成满足USB接口频率精度要求的48MHz时钟，以及最高133MHz的系统时钟。XOSC时钟也是时钟发生器的时钟源，因此在需要时可直接使用。

如用户已有精确时钟源，可直接将外部时钟输入XIN（即XI）引脚，且可禁用振荡电路。在该模式下，XIN的驱动频率最高可达50MHz。

如果用户希望在RP2040之外使用XOSC时钟，则必须通过clk_gpout时钟发生器将其路由至GPIO端口。不建议直接从XIN（又称XI）或XOUT（又称XO）获取信号。

图33. XOSC
概述

2.16.1.1 推荐晶体

为在典型工作温度范围内实现最佳性能与稳定性，建议使用Abracon ABM8-272-T3。您可直接向Abracon或其授权经销商采购该型号。Abracon ABM8-272-T3具有以下规格：

表257。关键晶体
规格

参数	最小值	典型值	最大值	单位	备注
中心频率	12.000	12.000	12.000	MHz	
工作模式	Fundamental-AT	Fundamental-AT	Fundamental-AT		
工作温度	-40		+85	°C	
存储温度	-55		+125	°C	
频率容差 (25°C)	-30		+30	ppm	
频率稳定性 (25°C)	-30		+30	ppm	
等效串联电阻 (R1)			50	Ω	
并联电容 (C0)			3.0	pF	
负载电容 (CL)	10	10	10	pF	
驱动功率		10	200	μW	
老化	-5		+5	ppm	@25±3°C, 第一年

参数	最小值	典型值	最大值	单位	备注
绝缘电阻	500			MΩ	@100Vdc±15V

即使使用规格相似的晶体振荡器，仍需在不同温度范围内测试电路以确保稳定性。

晶体振荡器由 VDDIO 电压供电。因此，Abracon 晶体及其特定阻尼电阻已经针对 3.3V 电压进行了调试。如果使用不同的 IO 电压，则需要重新调试。

对晶体参数的任何更改都可能导致连接于晶体电路的任何组件出现不稳定。

如果无法直接从 Abracon 或其经销商处采购推荐的晶体，请联系 applications@raspberrypi.com。

Raspberry Pi Pico 已专门针对 Abracon ABM8-272-T3 晶体的规格进行调试。有关在 RP2040 中使用晶体的示例，请参见 Raspberry Pi Pico 数据手册附录 B 中的 Raspberry Pi Pico 板原理图及设计文件。

2.16.2. 使用方法

XOSC 在芯片启动时处于禁用状态，RP2040 使用环形振荡器（ROSC）启动。要启动 XOSC，程序员必须设置 CTRL_ENABLE 寄存器。XOSC不可立即使用，因为振荡需要时间达到足够的幅度。该时间取决于所选晶体，但通常为几毫秒的量级。XOSC包含由STARTUP_DELAY寄存器控制的定时器，用于自动管理该过程，并在XOSC时钟可用时设置状态标志（STATUS_STABLE）。

2.16.3. 启动延迟

STARTUP_DELAY寄存器指定晶体必须检测到的时钟周期数，方可使用。

该数值以256的整数倍表示。该值由SDK中的 `xosc_init` 函数设置。1毫秒的默认值适用于RP2040参考设计（详见RP2040 硬件设计中的最小设计示例），该设计以12MHz频率运行XOSC。定时器到期后，STATUS_STABLE标志将被置位，指示 XOSC输出可用。

启动XOSC之前，程序员必须确保STARTUP_DELAY寄存器配置正确。所需的数值可通过以下公式计算：

$$(f_{Crystal} \times t_{Stable}) \div 256$$

因此，采用12MHz晶振和1ms等待时间，计算结果如下：

$$(12 \times 10^6 \cdot 1 \times 10^{-3}) \div 256 \approx 47$$

注意

该值向上取整至最接近的整数，因此实际等待时间略超过1ms。

2.16.4. XOSC 计数器

COUNT寄存器提供了一种管理短时软件延迟的方法。向COUNT寄存器写入数值后，将自动以XOSC频率开始倒计时至零。程序员只需轮询该寄存器，直到其值达到零。此方法优于在软件循环中使用NOP指令，因为它不依赖核心时钟频率、编译器或编译后代码的执行时间。

2.16.5. 休眠模式

在休眠模式（详见第2.11.3节）下，所有片上时钟均可暂停以节省功耗。此功能在电池供电的应用中尤为重要。RP2040通过外部事件（如GPIO引脚的边沿变化）或片上RTC的中断唤醒休眠模式。此配置须在进入休眠模式前完成。如果实时钟（RTC）用于触发唤醒，则必须由外部时钟源提供时钟。要进入休眠模式，程序员必须将所有内部时钟切换为由XOSC或ROSC驱动，并停止PLL。随后，必须向所选振荡器（XOSC或ROSC）的DORMANT寄存器写入特定的32位数值以停止振荡。退出休眠模式时，所选振荡器将重新启动。若选择XOSC，则频率更为精准，但因启动延迟，重启时间较长（RP2040参考设计中超过1ms，详见《RP2040硬件设计》中的最简设计示例）。若选择ROSC，频率较不精准，但启动时间极短（约1μs）。

注意

进入休眠模式前必须停止PLL。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c 第56至63行

```
56 void xosc_dormant(void) {
57     // 警告：此操作将停止xosc，直至通过irq唤醒
58     xosc_hw->dormant = XOSC_DORMANT_VALUE_DORMANT;
59     // 唤醒后，等待其稳定
60     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
61         tight_loop_contents();
62     }
63 }
```

警告

若在进入休眠模式前未配置IRQ，则XOSC或ROSC将无法重新启动。

有关使用XOSC休眠模式的完整示例，请参见第2.11.5.2节。

2.16.6. 程程序员模型

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/xosc.h 第27至59行

```
27 typedef struct {
28     _REG_(XOSC_CTRL_OFFSET) // XOSC_CTRL
29     // 晶体振荡器控制
30     // 0x00ffff000 [23:12] ENABLE      (-) 上电时，该字段初始化为 DISABLE，并且
31     // 该...
32     // 0x00000fff [11:0]   FREQ_RANGE   (-) 频率范围
33     io_rw_32 ctrl;
34
35     _REG_(XOSC_STATUS_OFFSET) // XOSC_STATUS
36     // 晶体振荡器状态
37     // 0x80000000 [31]      STABLE       (0) 振荡器正在运行且处于稳定状态
38 // 0x01000000 [24] BADWRITE (0) 向 CTRL_ENABLE 或...
39
40     // 0x00001000 [12]      ENABLED      (-) 振荡器已启用，但不一定正在运行
41     // 并且...
42     // 0x00000003 [1:0]      FREQ_RANGE   (-) 当前频率范围设置，始终读取为 0
43     io_rw_32 status;
44 }
```

```

42     _REG_(XOSC_DORMANT_OFFSET) // XOSC_DORMANT
43     // 晶体振荡器暂停控制
44     // 0xffffffff [31:0] DORMANT      (-) 用于通过暂停XOSC以节省功耗 +
45     io_rw_32 dormant;
46
47     _REG_(XOSC_STARTUP_OFFSET) // XOSC_STARTUP
48     // 控制启动延迟
49     // 0x00100000 [20]    X4          (-) 将startup_delay乘以4
50     // 0x00003fff [13:0]  DELAY       (-) 以256*xtal_period为单位的倍数
51     io_rw_32 startup;
52
53     uint32_t _pad0[3];
54
55     _REG_(XOSC_COUNT_OFFSET) // XOSC_COUNT
56     // 以XOSC频率运行的递减计数器，计数至零后停止。
57     // 0x000000ff [7:0]    COUNT      (0x00)
58     io_rw_32 count;
59 } xosc_hw_t;

```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c 第29至43行

```

29 void xosc_init(void) {
30     // 假设输入频率为1-15 MHz，已在前文确认。
31     xosc_hw->ctrl = XOSC_CTRL_FREQ_RANGE_VALUE_1_15MHZ;
32
33     // 设置xosc启动延迟
34     xosc_hw->startup = STARTUP_DELAY;
35
36     // 设置使能位，频率范围及启动延迟已配置完成
37     hw_set_bits(&xosc_hw->ctrl, XOSC_CTRL_ENABLE_VALUE_ENABLE << XOSC_CTRL_ENABLE_LSB);
38
39     // 等待XOSC稳定
40     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
41         tight_loop_contents();
42     }
43 }

```

2.16.7. 寄存器列表

XOSC寄存器起始基址为 **0x40024000**（在SDK中定义为XOSC_BASE）。

表258.
XOSC寄存器列表

偏移量	名称	说明
0x00	CTRL	晶体振荡器控制
0x04	状态	晶体振荡器状态
0x08	休眠	晶体振荡器暂停控制
0x0c	启动	控制启动延迟
0x1c	计数	一个以XOSC频率运行的递减计数器，计数至零后停止。

XOSC: CTRL寄存器

偏移: 0x00

描述

晶体振荡器控制

表259.
CTRL寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	启用: 上电时, 该字段初始化为禁用, 芯片运行于ROSC。 若芯片后来被配置为由XOSC供电, 则将此字段设为禁用可能导致芯片锁死。如有此风险, 建议将clk_ref运行于ROSC, 并启用clk_sys的RESUS功能。 该12位代码旨在防止意外写入。 无效的设置将启用振荡器。	读写	-
	枚举值:		
	0xd1e → DISABLE		
	0xfb → ENABLE		
11:0	FREQ_RANGE: 频率范围。此值重置为0xAA0, 且不可更改。	读写	-
	枚举值:		
	0xaa0 → 1_15MHZ		
	0xaa1 → RESERVED_1		
	0xaa2 → RESERVED_2		
	0xaa3 → RESERVED_3		

XOSC: STATUS寄存器

偏移: 0x04

描述

晶体振荡器状态

表260. STATUS
寄存器

位	描述	类型	复位值
31	STABLE: 振荡器正在运行且稳定	只读	0x0
30:25	保留。	-	-
24	BADWRITE: 已向CTRL_ENABLE、CTRL_FREQ_RANGE或DORMANT写入无效值	WC	0x0
23:13	保留。	-	-
12	ENABLED: 振荡器已启用, 但不一定正在运行且稳定, 复位值为0	只读	-
11:2	保留。	-	-
1:0	FREQ_RANGE: 当前频率范围设置, 始终读取为0	只读	-
	枚举值:		
	0x0 → 1_15MHZ		
	0x1 → RESERVED_1		
	0x2 → RESERVED_2		
	0x3 → RESERVED_3		

XOSC: 休眠模式寄存器

偏移量: 0x08

说明

晶体振荡器暂停控制

表 261. DORMANT
寄存器

位	描述	类型	复位值
31:0	此寄存器用于通过暂停XOSC以节省功耗 上电时该字段初始化为WAKE 无效写入同样会选择WAKE 警告：请选择休眠模式前必须停止PLL 警告：请选择休眠模式前须先设置中断请求	读写	-
	枚举值：		
	0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

XOSC: 启动寄存器

偏移: 0x0c

说明

控制启动延迟

表 262. STARTUP
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	X4: 将startup_delay乘以4。考虑到延迟可直接编程，用户对此价值有限。 。	读写	0x0
19:14	保留。	-	-
13:0	DELAY: 以256*xtal_period的倍数计。复位值0xc4对应约50,000个周期。	读写	0x00c4

XOSC: COUNT寄存器

偏移量: 0x1c

表 263. COUNT
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	一个以xosc频率运行的递减计数器，计数至零后停止。 要启动计数器，请写入非零值。 可用于设置时间敏感硬件时的短时软件延迟。	读写	0x00

2.17. 环形振荡器 (ROSC)

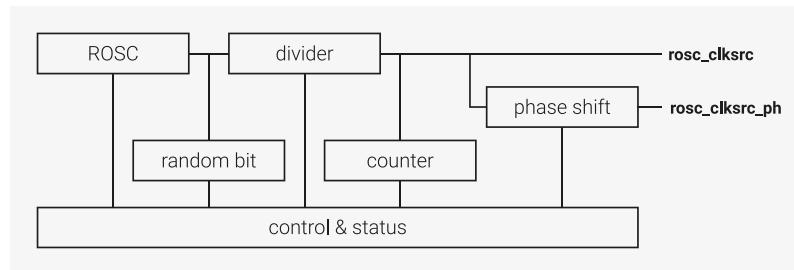
2.17.1. 概述

环形振荡器 (ROSC) 为由反相器环构成的片上振荡器。其无需外部元件，并在RP2040上电时自动启动，启动期间为核心提供时钟。

ROSC频率可编程，并可直接向核心提供高速时钟，但其频率随工艺、供电电压及温度（PVT）变化，因此无法为需精准频率的组件（如RTC、USB及ADC）提供时钟。第2.15节讨论了频率波动的缓解方案，仅适用于超低功耗设计。对大多数要求精确时钟频率的应用，建议切换至XOSC和PLL。ROSC启动时以标称6.5MHz运行，保证频率范围为1.8MHz至12MHz。

芯片启动后，程序员可选择继续使用ROSC运行并提高其频率，或启动晶体振荡器（XOSC）及PLL。系统时钟切换至XOSC后，ROSC可被禁用。各振荡器各具优势，程序员可根据应用需求切换使用，以实现最佳解决方案。

图34. ROSC
概述。



2.17.2. ROSC/XOSC 折衷

ROSC的优势在于其灵活性及低功耗。此外，使用ROSC提供时钟时，无需任何内部或外部组件。其频率可编程，可用于提供快速内核时钟而无需启动PLL，且可通过时钟发生器（见第2.15节）分频产生较慢的外设时钟。ROSC可立即启动，且即时响应频率控制。进入及退出休眠状态（见第2.11.3节）时，ROSC将保持频率设置。但用户须注意，由于电源电压和芯片温度变化，退出休眠状态时频率可能会漂移。

ROSC 的缺点在于其频率随着 PVT（工艺、电压和温度）的变化而变化，这使其不适合用于生成精确时钟或对软件执行时序要求严格的应用。然而，PVT 频率的变化可以被利用，实现自动频率调整，以最大化性能。相关内容详见第 2.15 节。

XOSC 的唯一优势在于其频率的准确性，这对许多应用而言是不可替代的要求。

XOSC 的缺点包括需要外部元件（如晶体）、较高的功耗、启动时间较长（>1ms）以及固定且较低的频率。必须使用 PLL 以产生更高频率的时钟信号。

PLL 功耗更高，启动及频率调整所需时间较长。退出休眠模式比 ROSC 慢得多，因需重新启动 XOSC 并重新配置 PLL。

2.17.3. 频率调整

ROSC 由 8 级组成，每级均具备可编程驱动能力。控制频率的方法有两种。

频率范围控制ROSC环路中的级数，FREQA和FREQB寄存器则控制各级的驱动强度。

通过写入FREQ_RANGE寄存器可以改变频率范围，该寄存器决定ROSC环路中的级数。默认情况下，LOW范围包含8级（第0至7级），MEDIUM为6级（第2至7级），HIGH为4级（第4至7级），TOOHIGH为2级（第6至7级）。建议逐步调整FREQ_RANGE，每次调整一步，直至达到所需的频率范围。增加频率范围时，ROSC输出不会出现毛刺，因此输出时钟可以持续使用。但在降低频率范围时，情况则不同。对于由ROSC时钟提供时钟信号的模块，必须选择备用时钟源，或在切换期间将其保持复位状态。该行为尚未完全描述，但中等范围大约是低范围的1.33倍，高范围将是低范围的2倍。

低范围和过高范围将是低范围的4倍。“过高范围”称谓恰当。不应使用过高范围，因为ROSC的内部逻辑无法在该频率下正常运行。

FREQA与FREQB寄存器控制ROSC环路各阶段的驱动强度。增加驱动强度可缩短阶段延迟，提升振荡频率。每个阶段设有3个位的驱动强度控制位。每个位启用额外驱动，因此每个阶段共有4个驱动强度设定，取决于已置位数量：0为默认，1为双倍驱动，2为三倍驱动，3为四倍驱动。开启额外驱动对频率的影响非线性，第二个位的影响小于第一个，依此类推。

为确保过渡平稳，建议每次仅修改一个驱动强度位。当使用FREQ_RANGE缩短ROSC环路时，旁路的阶段仍会传递信号，因此其驱动强度必须设置为至少与环路中阶段的最低驱动强度相等。此操作不会影响振荡频率。

2.17.4. ROSC 分频器

ROSC频率过高，无法直接使用，因此通过由DIV寄存器控制的整数分频器进行分频。DIV可在ROSC运行时更改，输出时钟频率将无毛刺地变化。默认分频值为16，保证芯片启动时输出时钟频率处于1.8至12MHz范围内。

分频器设有两个输出，rosclksrc和rosclksrc_ph，后者为前者的相位移版本。此功能主要用于产品开发阶段，若PHASE寄存器保持默认状态，两个输出信号将完全相同。

2.17.5. 随机数生成器

如果系统时钟由XOSC和/或PLL驱动，则ROSC可用于生成随机数。

只需启用ROSC并读取RANDOMBIT寄存器，即可获得1位随机数，重复读取n次以获得n位随机数值。此方法无法满足安全系统对随机性的要求，因其可能被攻破，但在非关键应用中仍可使用。若核心运行于ROSC时钟，读取的值将不具随机性，因为寄存器读取的时序与ROSC的相位相关。

2.17.6. ROSC 计数器

COUNT寄存器提供了一种管理短时软件延迟的方法。向COUNT寄存器写入数值后，将自动以ROSC频率开始倒计时至零。程序员只需轮询该寄存器，直到其值达到零。此方法优于在软件循环中使用NOP指令，因为它不依赖核心时钟频率、编译器或编译后代码的执行时间。

2.17.7. 休眠模式

在休眠模式（详见第2.11.3节）下，所有片上时钟均可暂停以节省功耗。此功能在电池供电的应用中尤为重要。RP2040通过外部事件（如GPIO引脚的边沿变化）或片上RTC的中断唤醒休眠模式。此配置须在进入休眠模式前完成。如果实时钟（RTC）用于触发唤醒，则必须由外部时钟源提供时钟。要进入休眠模式，程序员必须将所有内部时钟切换为由XOSC或ROSC驱动，并停止PLL。随后，必须向所选振荡器（XOSC或ROSC）的DORMANT寄存器写入特定的32位数值以停止振荡。退出休眠模式时，所选振荡器将重新启动。若选择XOSC，则频率更为精准，但因启动延迟，重启时间较长（RP2040参考设计中超过1ms，详见《RP2040硬件设计》中的最简设计示例）。若选择ROSC，频率较不精准，但启动时间极短（约1μs）。

Pico附加资源: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/hardware_osc/osc.c 第56至61行

```

56 void rosc_set_dormant(void) {
57     // 警告：此操作将停止 rosc，直至通过中断唤醒
58     rosc_write(&rosc_hw->dormant, ROSC_DORMANT_VALUE_DORMANT);
59     // 等待唤醒后稳定
60     while(!(rosc_hw->status & ROSC_STATUS_STABLE_BITS));
61 }
```

● 警告

若在进入休眠模式前未配置IRQ，ROSC将无法重新启动。

有关休眠模式的部分示例，详见第2.11.5.2节。

2.17.8. 寄存器列表

ROSC寄存器起始地址为 **0x40060000** (SDK中定义为ROSC_BASE)。

表264.
ROSC寄存器列表

偏移量	名称	说明
0x00	CTRL	环振荡器控制
0x04	FREQA	环振荡器频率控制A
0x08	FREQB	环振荡器频率控制B
0x0c	休眠	环振荡器暂停控制
0x10	DIV	控制输出分频器
0x14	PHASE	控制相移输出
0x18	状态	环振荡器状态
0x1c	RANDOMBIT	返回1位随机值
0x20	计数	一个以ROSC频率运行的向下计数器，计数至零后停止。

ROSC: CTRL寄存器

偏移: 0x00

描述

环振荡器控制

表265. CTRL
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	ENABLE: 上电时此字段初始化为ENABLE 在将此字段设置为DISABLE之前，必须将系统时钟切换至其他时钟源，否则芯片将锁死 该12位代码旨在防止意外写入。 无效的设置将启用振荡器。	读写	-
	枚举值：		
	0xd1e → DISABLE		
	0xfb → ENABLE		

位	描述	类型	复位值
11:0	FREQ_RANGE: 控制ROSC环中的延迟阶段数 LOW使用第0至7阶段 MEDIUM使用第2至7阶段 HIGH使用第4至7阶段 TOOHIGH使用第6至7阶段，因其频率超过设计规格，故不应使用 时钟输出在逐级向上调整范围时不会产生毛刺 时钟输出在调整范围向下时会产生毛刺 注：此处数值采用格雷码，这也是HIGH排在TOOHIGH之前的原因	读写	0xaa0
	枚举值：		
	0xfa4 → LOW		
	0xfa5 → MEDIUM		
	0xfa7 → 高		
	0xfa6 → 过高		

ROSC: FREQA 寄存器

偏移量: 0x04

说明

FREQA 和 FREQB 寄存器通过控制各阶段的驱动强度来调节频率。驱动强度分为4级，由置位的位数决定。置位位数增加会增强驱动强度并提升振荡频率。

- 0 位置位为默认驱动强度
- 1 位置位使驱动强度加倍
- 2 位置位使驱动强度增加三倍
- 3 位置位使驱动强度增加四倍

表 266. FREQA 寄存器

位	描述	类型	复位值
31:16	PASSWD: 设置为 0x9696 以应用设置 该字段的任何其他数值均将使所有驱动强度归零	读写	0x0000
	枚举值：		
	0x9696 → 通过		
15	保留。	-	-
14:12	DS3: 第3阶段驱动强度	读写	0x0
11	保留。	-	-
10:8	DS2: 第2阶段驱动强度	读写	0x0
7	保留。	-	-
6:4	DS1: 第1阶段驱动强度	读写	0x0
3	保留。	-	-
2:0	DS0: 第0阶段驱动强度	读写	0x0

ROSC: FREQB 寄存器

偏移: 0x08

描述

详见 freqa 寄存器的详细描述

表 267。FREQB
寄存器

位	描述	类型	复位值
31:16	PASSWD : 设置为 0x9696 以应用设置 该字段的任何其他数值均将使所有驱动强度归零	读写	0x0000
	枚举值:		
	0x9696 → 通过		
15	保留。	-	-
14:12	DS7 : 第 7 级驱动强度	读写	0x0
11	保留。	-	-
10:8	DS6 : 第 6 级驱动强度	读写	0x0
7	保留。	-	-
6:4	DS5 : 第 5 级驱动强度	读写	0x0
3	保留。	-	-
2:0	DS4 : 第 4 级驱动强度	读写	0x0

ROSC：休眠模式寄存器

偏移: 0x0c

说明

环振荡器暂停控制

表 268。休眠模式
寄存器

位	描述	类型	复位值
31:0	该寄存器用于通过暂停 ROSC 来节省功耗 上电时该字段初始化为 WAKE 无效写入同样会选择 WAKE 警告: 请在选择休眠模式之前设置 irq	读写	-
	枚举值:		
	0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

ROSC：DIV 寄存器

偏移: 0x10

说明

控制输出分频器

表 269。DIV
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	设置为 $0xaaa + div$, 其中 $div = 0$ 表示除以 32 $div = 1-31$ 表示除以 div 其他值均设定 $div = 31$ 该寄存器复位时, $div = 16$	读写	-
	枚举值:		

位	描述	类型	复位值
	0xaa0 → 通过		

ROSC: PHASE 寄存器

偏移: 0x14

说明

控制相移输出

表270. PHASE 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:4	PASSWD : 设置为0xaa 其他任意值启用输出且shift=0	读写	0x00
3	ENABLE : 启用相位偏移输出 该项支持动态修改	读写	0x1
2	FLIP : 反转相位偏移输出 当div=1时，将忽略此项	读写	0x0
1:0	SHIFT : 对相位偏移输出进行SHIFT输入时钟的相位移 该项支持动态修改 设置div=1前，须先将其设为0	读写	0x0

ROSC: STATUS 寄存器

偏移: 0x18

描述

环振荡器状态

表271. STATUS 寄存器

位	描述	类型	复位值
31	STABLE : 振荡器正在运行且稳定	只读	0x0
30:25	保留。	-	-
24	BADWRITE : 向CTRL_ENABLE、CTRL_FREQ_RANGE、FREQA、 FREQB、DIV、PHASE或DORMANT寄存器写入了无效值	WC	0x0
23:17	保留。	-	-
16	DIV_RUNNING : 后置分频器正在运行 该值复位为0，但芯片启动期间会变为1	只读	-
15:13	保留。	-	-
12	ENABLED : 振荡器已启用，但不一定正在运行或处于稳定状态 该值复位为0，但芯片启动期间会变为1	只读	-
11:0	保留。	-	-

ROSC: RANDOMBIT 寄存器

偏移量: 0x1c

表272。
RANDOMBIT 寄存器

位	描述	类型	复位值
31:1	保留。	-	-

位	描述	类型	复位值
0	该寄存器仅读取振荡器输出状态，因此当环形振荡器停止或以总线频率的谐波运行时，随机性将受损。	只读	0x1

ROSC: COUNT 寄存器

偏移量: 0x20

表 273。COUNT
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	一个以ROSC频率运行的向下计数器，计数至零后停止。 要启动计数器，请写入非零值。 可用于设置时间敏感硬件时的短时软件延迟。	读写	0x00

2.18. PLL

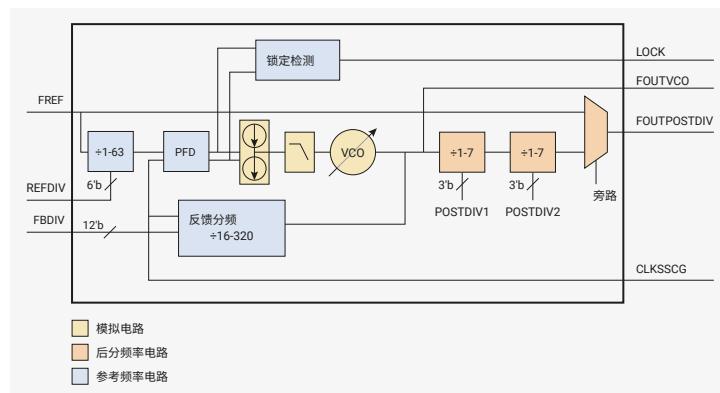
2.18.1. 概述

PLL 设计用于接收参考时钟，并通过带有反馈环路的压控振荡器（VCO）进行倍频。VCO 必须在高频率范围运行（750 至 1600 MHz），因此设有两个后置分频器，可在将 VCO 频率分配给芯片上的时钟发生器之前对其进行分频。

RP2040 内置两个 PLL，如下所示：

- **pll_sys** - 用于生成最高 133 MHz 的系统时钟
- **pll_usb** - 用于生成 48 MHz 的 USB 参考时钟

图35。两个PLL 中，REF（参考）输入均连接至晶体振荡器的X输入端。PL 包含一个VC O，通过反馈环路（相位频率检测器和环路滤波器）将其锁定至参考时钟的恒定倍数。该设计能够合成极高的频率，随后可由后分频器进行降频。



2.18.2. PLL 参数计算

配置PLL时，必须了解参考时钟频率，RP2040中该频率直接取自晶体振荡器。该参考时钟通常为12MHz晶体，以兼容RP 2040的USB引导只读存储器。PLL的最终输出频率FOUTPOSTDIV可按公式计算： $(\text{REF} / \text{REFDIV}) \times \text{FB DIV} / (\text{POSTDIV1} \times \text{POSTDIV2})$ 。基于所需的输出频率，必须依照以下PLL设计约束选择PLL参数：

- 最小参考频率 (**REF / REFDIV**) 为5MHz

- 振荡器频率 (**FOUTVCO**) 须处于 $750\text{MHz} \rightarrow 1600\text{MHz}$ 范围内
- 反馈分频器 (**FBDIV**) 须处于 $16 \rightarrow 320$ 范围内
- 后置分频器 **POSTDIV1**和 **POSTDIV2**须处于 $1 \rightarrow 7$ 范围内
- 最大输入频率 (**FREF / REFDIV**) 为VCO频率除以16, 此因最小反馈分频器限制

此外, 芯片时钟生成器 (连接至**FOUTPOSTDIV**) 的最大频率亦须被遵守。系统PLL为 133MHz , USB PLL为 48MHz 。

注意

RP2040上的晶体振荡器设计适用于5至 15MHz 范围的晶体, 因此通常 **REFDIV**应设为1。如果应用电路直接驱动更高速率参考信号进入XI输入, 且希望获得较低的VCO频率, 则可以增大参考分频, 以保持PLL输入处于适宜范围。

提示

当 **POSTDIV1**和 **POSTDIV2**需要赋予不同数值时, 建议将较大数值赋予 **POSTDIV1**, 以实现更低的功耗。

在RP2040参考设计中 (参见《RP2040硬件设计》中的极简设计示例), 其将一枚 12MHz 晶体连接至晶体振荡器, 这意味着最低可实现且合法的VCO频率为 $12\text{MHz} \times 63 = 756\text{MHz}$, 且最大VCO频率为 $12\text{MHz} \times 133 = 1596\text{MHz}$, 因此 **FBDIV**必须保持在 $63 \rightarrow 133$ 范围内。例如, 将 **FBDIV**设定为100将合成出 1200MHz 的VCO频率。若 **POSTDIV1**设为6, **POSTDIV2**设为2, 则总体分频为12, 从而在PLL最终输出端产生稳定的 100MHz 信号。

2.18.2.1. 抖动与功耗

通常存在若干组PLL配置参数能够实现或非常接近所需的输出频率。程序员需决定是优先降低PLL功耗, 还是降低抖动—即PLL输出时钟周期之间的变化。这对系统稳定性不构成影响, 因为RP2040的数字逻辑设计中预留了系统时钟最坏情况抖动的裕量, 但在音频、视频应用或依据规范进行数据传输时, 通常需要极高精度的时钟。例如, USB规范规定了允许的最大抖动值。

抖动通过使VCO以尽可能高的频率运行, 从而采用更高的后分频值得以最小化。

例如, 1500MHz 的VCO经过6和2的连续分频后得到 125MHz 。为降低功耗, VCO频率应尽可能降低。例如: $750\text{MHz} \text{ VCO} / 6 / 1 = 125\text{MHz}$ 。

另一个需要考虑的问题是, 通过将输出频率调整至接近输入频率的有理数倍, 可能实现更低的VCO频率。实际上, 任何允许的VCO频率或除数组合可能无法精确达到所需的确切频率。

SDK提供了一个Python脚本, 用以搜索满足目标输出频率的最佳VCO和后置分频选项:

SDK链接: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/scripts/vcocalc.py

```

1 #!/usr/bin/env python3
2
3 import argparse
4 import sys
5
6 #固定硬件参数
7 fbdiv_range = range(16, 320 + 1)
8 postdiv_range = range(1, 7 + 1)
9 ref_min = 5
10 refdiv_min = 1
11 refdiv_max = 63

```

```

12
13 def validRefdiv(string):
14     if ((int(string) < refdiv_min) or (int(string) > refdiv_max)):
15         raise ValueError("REFDIV 必须位于 {} 至 {} 范围内".format(refdiv_min, refdiv_max))
16     return int(string)
17
18 parser = argparse.ArgumentParser(description="PLL 参数计算器")
19 parser.add_argument("--input", "-i", default=12, help="输入 (参考) 频率, 默认 12 MHz", type=float)
20 parser.add_argument("--ref-min", default=5, help="覆盖最小参考频率, 默认 5 MHz", type=float)
21 parser.add_argument("--vco-max", default=1600, help="覆盖最大 VCO 频率, 默认 1600 MHz", type=float)
22 parser.add_argument("--vco-min", default=750, help="覆盖最小 VCO 频率, 默认 750 MHz", type=float)
23 parser.add_argument("--cmake", action="store_true", help="打印CMake代码片段以将所选PLL参数应用于您的程序")
24 parser.add_argument("--cmake-only", action="store_true", help="功能同--cmake, 但仅输出CMake内容, 不打印其他信息")
25 parser.add_argument("--cmake-executable-name", default=<program>, help="设置生成CMake输出时使用的可执行文件名称")
26 parser.add_argument("--lock-refdiv", help="将REFDIV锁定为范围{} 至{}内的指定数值".format(refdiv_min, refdiv_max), type=validRefdiv)
27 parser.add_argument("--low-vco", "-l", action="store_true", help="在可能时使用较低的VCO频率, 以降低功耗, 但会增加抖动")
28 parser.add_argument("output", help="输出频率 (MHz)", type=float)
29 args = parser.parse_args()
30
31 refdiv_range = range(refdiv_min, max(refdiv_min, min(refdiv_max, int(args.input / args.ref_min))) + 1)
32 if args.lock_refdiv:
33     print("锁定REFDIV为", args.lock_refdiv)
34     refdiv_range = [args.lock_refdiv]
35
36 best = (0, 0, 0, 0, 0, 0)
37 best_margin = args.output
38
39 for refdiv in refdiv_range:
40     for fbdv in fbdv_range:
41         vco = args.input / refdiv * fbdv
42         if vco < args.vco_min or vco > args.vco_max:
43             continue
44         # pd1 是内层循环, 因此优先更高的 pd1:pd2 比率
45         for pd2 in postdiv_range:
46             for pd1 in postdiv_range:
47                 out = vco / pd1 / pd2
48                 margin = abs(out - args.output)
49                 vco_is_better = vco < best[5] if args.low_vco else vco > best[5]
50                 if ((vco * 1000) % (pd1 * pd2)):
51                     continue
52                 if margin < best_margin or (abs(margin - best_margin) < 1e-9 and
53                     vco_is_better):
54                     best = (out, fbdv, pd1, pd2, refdiv, vco)
55                     best_margin = margin
56
57 best_out, best_fbdv, best_pd1, best_pd2, best_refdiv, best_vco = best
58
59 if best[0] > 0:
60     cmake_output = \
61     """target_compile_definitions({args.cmake_executable_name} PRIVATE
62     PLL_SYS_REFDIV={best_refdiv}
63     PLL_SYS_VCO_FREQ_HZ={int((args.input * 1_000_000) / best_refdiv * best_fbdv)}
64     PLL_SYS_POSTDIV1={best_pd1}"""

```

```

64     PLL_SYS_POSTDIV2={best_pd2}
65     SYS_CLK_HZ=(int((args.input * 1_000_000) / (best_refdiv * best_pd1 * best_pd2) *
66     best_fbddiv)}
66 )
67 """
68     if not args.cmake_only:
69         print("请求频率: {} MHz".format(args.output))
70         print("实现频率: {} MHz".format(best_out))
71         print("REFDIV:    {}".format(best_refdiv))
72         print("FBDIV:    {} (VCO = {} MHz)".format(best_fbddiv, args.input / best_refdiv *
73             best_fbddiv))
73         print("PD1:      {}".format(best_pd1))
74         print("PD2:      {}".format(best_pd2))
75         if best_refdiv != 1:
76             print(
77                 "\n这需要非默认的 REFDIV 值。\\n"
78                 "请将以下内容添加至您的 CMakeLists.txt 以应用该 REFDIV:\\n"
79             )
80     elif args.cmake 或 args.cmake_only:
81         print("")
82     if args.cmake 或 args.cmake_only 或 best_refdiv != 1:
83         print(cmake_output)
84 else:
85     sys.exit("未找到解决方案")

```

给定输入和输出频率，本脚本将寻找最佳PLL参数组合，以尽可能接近目标频率。当存在多个同样优良的组合时，返回能产生最高VCO频率的参数，以保证最佳输出稳定性。使用 `-l` 或 `--low-vco` 选项将优先选择较低频率，以降低功耗。

此处请求一个48 MHz输出：

```

$ ./vcocalc.py 48
请求频率: 48.0 MHz
实现频率: 48.0 MHz
FBDIV: 120 (VCO = 1440 MHz)
PD1: 6
PD2: 5

```

若可能，希望以较低VCO频率输出48 MHz：

```

$ ./vcocalc.py -l 48
请求频率: 48.0 MHz
实现频率: 48.0 MHz
FBDIV: 64 (VCO = 768 MHz)
PD1: 4
PD2: 4

```

对于12MHz输入信号的125MHz系统时钟，最低VCO频率相当高。

```

$ ./vcocalc.py -l 125
请求频率: 125.0 MHz
实际频率: 125.0 MHz
FBDIV: 125 (VCO = 1500 MHz)
PD1: 6

```

PD2 : 2

我们可以将搜索范围限制在较低的VCO频率内，使脚本考虑更宽松的频率匹配。请注意，尽管750MHz的VCO频率在此处较为理想，但由于无法通过将12MHz输入信号乘以整数得到精确的750MHz，因此之前的调用返回了较高的VCO频率。

```
$ ./vcocalc.py -l 125 --vco-max 800
请求频率: 125.0 MHz
实际频率: 126.0 MHz
FB DIV: 63 (VCO = 756 MHz)
PD1: 6
PD2: 1
```

126MHz的系统时钟可能是对目标125MHz的可接受偏差，且在PLL中生成此时钟时功耗较低。

2.18.3. 配置

SDK使用以下PLL设置：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware/clocks.h 第143至164行

```
143 // RP 系列微控制器中有两个PLL：
144 // 1. 'SYS PLL' 用于生成系统时钟，频率由 `SYS_CLK_KHZ` 定义。
145 // 2. 'USB PLL' 用于生成USB时钟，频率由 `USB_CLK_KHZ` 定义。
146 //
147 // 两个PLL 均直接使用晶体振荡器的输出作为其参考频率输入；
// PLL 的参考
148 // 频率不能被时钟模块中的分频器降低。晶体振荡器频率由 `XOSC_HZ` 定义（或 `XOSC_KHZ` 或 `XOSC_MHZ`）。

150 //
151 // 系统默认定义适用于上述频率，晶体频率为12MHz
152 // 晶体频率。若需不同频率，必须在
153 // 板级配置文件中定义，并同时调整PLL 设置。
154 // 如果更改任一频率，请使用 `vcocalc.py` 检查并计算新的PLL 设置。

155 //
156 // RP2040默认PLL配置：
157 //           REF      FB DIV VCO          POSTDIV
158 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 6 / 2 = 125MHz
159 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 =        48MHz
160 //
161 // RP2350默认PLL配置：
162 //           REF      FB DIV VCO          POSTDIV
163 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 5 / 2 = 150MHz
164 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 =        48MHz
```

SDK 中的 `pll_init` 函数（我们将在下文详细审查）断言所有这些条件均成立后，方尝试配置 PLL。

SDK 将 PLL 控制寄存器定义为结构体。随后，将其映射至每个 PLL 实例的内存中。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h 第27行至53行

```

27 typedef struct {
28     _REG_(PLL_CS_OFFSET) // PLL_CS
29     // 控制与状态
30     // 0x80000000 [31]      LOCK        (0) PLL 已锁定
31     // 0x00000100 [8]       BYPASS      (0) 将参考时钟直接传递至输出
32     // 0x0000003f [5:0]     REFDIV      (0x01) 除以PLL输入参考时钟
33     io_rw_32 cs;
34
35     _REG_(PLL_PWR_OFFSET) // PLL_PWR
36     // 控制PLL电源模式
37     // 0x00000020 [5]       VCOPD      (1) PLL VCO电源关闭 +
38     // 0x00000008 [3]       POSTDIVPD    (1) PLL 后置分频器电源关闭 +
39     // 0x00000004 [2]       DSMPD      (1) PLL DSM电源关闭 +
40     // 0x00000001 [0]       PD          (1) PLL 电源关闭 +
41     io_rw_32 pwr;
42
43     _REG_(PLL_FBDIV_INT_OFFSET) // PLL_FBDIV_INT
44     // 反馈分频器
45     // 0x00000fff [11:0]    FB DIV INT   (0x000) 详见控制寄存器说明的限制
46     io_rw_32 fbdv_int;
47
48     _REG_(PLL_PRIM_OFFSET) // PLL_PRIM
49     // 控制主输出的PLL 后置分频器
50     // 0x00070000 [18:16] POSTDIV1      (0x7) 分频系数为1至7
51     // 0x00007000 [14:12] POSTDIV2      (0x7) 分频系数为1至7
52     io_rw_32 prim;
53 } pll_hw_t;
```

SDK 定义了 `pll_init` 函数，用于配置或重新配置 PLL。该函数首先清除 PLL 中之前的电源状态，随后计算适当的反馈分频值。通过断言检查这些值是否满足上述约束条件。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pll/pll.c 第13行至21行

```

13 void pll_init(PLL pll, uint refdiv, uint vco_freq, uint post_div1, uint post_div2) {
14     uint32_t ref_freq = XOSC_HZ / refdiv;
15
16     // 检查VCO频率是否位于可接受范围内
17     assert(vco_freq >= PICO_PLL_VCO_MIN_FREQ_HZ && vco_freq <= PICO_PLL_VCO_MAX_FREQ_HZ);
18
19     // 我们将参考时钟乘以多少以获得VCO频率
20 // (寄存器命名为div，因为你将VCO输出除以该值并与参考时钟比较)
21     uint32_t fbdiv = vco_freq / ref_freq;
```

PLL的编程顺序如下：

- 编程参考时钟分频器（RP2040中为1倍分频）
- 编程反馈分频器
- 开启主电源和VCO
- 等待VCO锁定（即保持输出频率稳定）
- 设置后级分频器并启用

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pll/pll.c 第42至69行

```

42     如果 ((pll->cs & PLL_CS_LOCK_BITS) &&
43         (refdiv == (pll->cs & PLL_CS_REFDIV_BITS)) &&
44         (fbdiv == (pll->fbdiv_int & PLL_FBDIV_INT_BITS)) &&
45         (pdiv == (pll->prim & (PLL_PRIM_POSTDIV1_BITS | PLL_PRIM_POSTDIV2_BITS))) {
46         // 不要干扰已正确配置且正在运行的PLL
47         返回;
48     }
49
50     reset_unreset_block_num_wait_blocking(PLL_RESET_NUM(pll));
51
52     // 在启动VCO之前加载与VCO相关的分频器
53     pll->cs = refdiv;
54     pll->fbdiv_int = fbdiv;
55
56     // 启动PLL
57     uint32_t power = PLL_PWR_PD_BITS | // 主电源
58                           PLL_PWR_VCOPD_BITS; // VCO电源
59
60     hw_clear_bits(&pll->pwr, power);
61
62     // 等待PLL锁定
63     while (!(pll->cs & PLL_CS_LOCK_BITS)) tight_loop_contents();
64
65     // 配置后置分频器
66     pll->prim = pdiv;
67
68     // 启用后置分频器
69     hw_clear_bits(&pll->pwr, PLL_PWR_POSTDIVPD_BITS);

```

请注意，VCO 先行启动，随后启动后置分频器，以确保 PLL 在 VCO 锁定期间不会输出杂讯时钟。

2.18.4. 寄存器列表

PLL_SYS 和 PLL_USB 寄存器的基地址分别为 `0x40028000` 和 `0x4002c000`（在 SDK 中分别定义为 `PLL_SYS_BASE` 和 `PLL_USB_BASE`）。

表 274. PLL
寄存器列表

偏移量	名称	说明
0x0	CS	控制与状态
0x4	PWR	控制 PLL 电源模式。
0x8	FBDIV_INT	反馈分频器
0xc	PRIM	控制主输出的 PLL 后置分频器

PLL：CS 寄存器

偏移量: 0x0

描述

控制与状态

通用约束:

参考时钟频率，最小值=5MHz，最大值=800MHz

反馈分频，最小值=16，最大值=320

VCO频率，最小值=750MHz，最大值=1600MHz

表275. CS寄存器

位	描述	类型	复位值
31	LOCK: PLL已锁定	只读	0x0
30:9	保留。	-	-
8	BYPASS: 直接传递参考时钟至输出，替代分频后的VCO信号。 VCO持续运行，用户可在参考时钟与分频VCO间切换，但切换时输出信号会出现毛刺。	读写	0x0
7:6	保留。	-	-
5:0	REFDIV: 对PLL输入参考时钟进行分频。 当div=0时，行为未定义。 PLL输出在refdiv更改期间不可预测，使用前请等待lock=1。	读写	0x01

PLL: PWR寄存器

偏移：0x4

说明

控制 PLL 电源模式。

表276. PWR 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	VCOPD: PLL VCO断电 在不需要PLL输出或bypass=1时，为节省功耗应置高。	读写	0x1
4	保留。	-	-
3	POSTDIVPD: PLL后级分频断电 在不需要PLL输出或bypass=1时，为节省功耗应置高。	读写	0x1
2	DSMPD: PLL DSM断电 将此设置为较低值不会产生任何效果。	读写	0x1
1	保留。	-	-
0	PD: PLL 电源关闭 当不需要 PLL 输出时，置为高电平以节省电能。	读写	0x1

PLL: FBDIV_INT 寄存器

偏移量：0x8

说明

反馈分频器

(注：该 PLL 不支持分数分频)

表 277. FBDIV_INT 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	有关限制条件，请参阅控制寄存器描述。	读写	0x000

PLL: PRIM 寄存器

偏移: 0xc

描述

控制主输出的 PLL 后置分频器

(注：该 PLL 不具有次级输出)
主输出由 VCO 除以 postdiv1*postdiv2 后驱动。

表 278. PRIM
寄存器

位	描述	类型	复位值
31:19	保留。	-	-
18:16	POSTDIV1 : 除以 1 至 7	读写	0x7
15	保留。	-	-
14:12	POSTDIV2 : 除以 1 至 7	读写	0x7
11:0	保留。	-	-

2.19. GPIO

2.19.1. 概述

RP2040 配备 36 个多功能通用输入输出 (GPIO) 引脚，分为两组。在典型应用中，QSPI 组的引脚（QSPI_SS、QSPI_SC_LK 及 QSPI_SD0 至 QSPI_SD3）用于从外部闪存设备执行代码，用户组（GPIO0 至 GPIO29）则供程序员使用。所有 GPIO 支持数字输入和输出，其中 GPIO26 至 GPIO29 还可作为芯片的模拟数字转换器（ADC）输入。

每个GPIO均可通过运行于处理器上的软件直接控制，或由其他若干功能模块进行控制。

用户GPIO组支持以下功能：

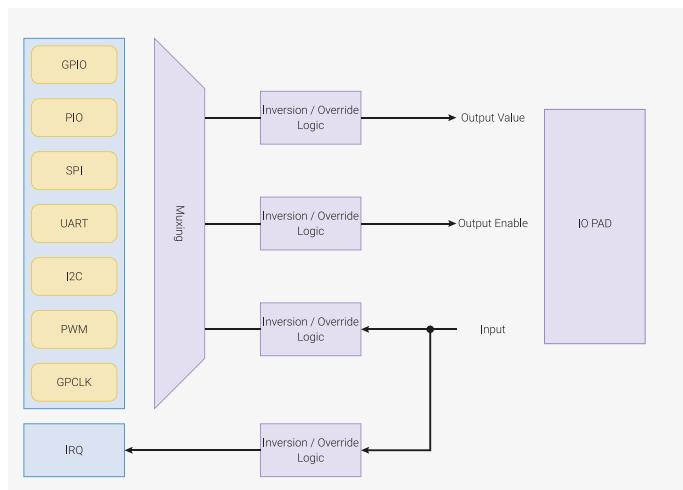
- 通过SIO（单周期IO）进行软件控制——第2.3.1.2节，“GPIO控制”
- 可编程IO（PIO）——第3章， *PIO*
- 2×SPI——第4.4节，“SPI”
- 2 ×UART——第4.2节，“UART”
- 2 ×I2C（两线串行接口）——第4.3节，“I2C”
- 8 ×双通道PWM——第4.5节，“PWM”
- 2 ×外部时钟输入——第2.15.2.3节，“外部时钟”
- 4 ×通用时钟输出——第2.15节，“时钟”
- 4 ×作为ADC输入——第4.9节，“ADC与温度传感器”
- USB VBUS管理——第4.1.2.10节，“VBUS控制”
- 外部中断请求，电平或边沿触发

QSPI 组支持以下功能：

- 通过SIO（单周期IO）进行软件控制——第2.3.1.2节，“GPIO控制”
- Flash 执行就地 (XIP) — 见第 2.6.3 节，“Flash”

示例 IO 的逻辑结构如图 36 所示。

图36。GPIO的逻辑结构。
每个GPIO可以由多个外设之一控制，或由SIO中的软件控制寄存器控制。
功能选择(FSEL)用于选择由哪个外设输出控制GPIO的方向和输出电平，和／或由哪个外设输入读取该GPIO的输入电平。这三个信号（输出电平、输出使能、输入电平）也可以通过GPIO控制寄存器进行反相，或强制置高或置低。



2.19.2. 功能选择

分配给每个GPIO的功能，通过向GPIO的CTRL寄存器中的FUNCSEL字段写入来选择。以GPIO0_CTRL为例。各IO上可用的功能详见表 279 和表 281。

表279。通用输入/输出(GPIO)用户组功能

GPIO	功能									
	F1	F2	F3	F4	F5	F6	F7	F8	F9	
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB 过流检测
1	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1			USB 总线电压检测
2	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
3	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1			USB 过流检测
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1			USB 总线电压检测
5	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
6	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1			USB 过流检测
7	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1			USB 总线电压检测
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
9	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1			USB 过流检测
10	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1			USB 总线电压检测
11	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1			USB 过流检测
13	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1			USB 总线电压检测
14	SPI1 SCK	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM7 A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
15	SPI1 TX	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM7 B	SIO	PIO0	PIO1			USB 过流检测
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB 总线电压检测
17	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
18	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1			USB 过流检测
19	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1			USB 总线电压检测
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0		USB 总线使能 (VBUS EN)

功能									
21	SPI0 片选信号 (CSn)	UART1 RX	I ₂ C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB 过流检测
22	SPI0 时钟信号 (SCK)	UART1 CTS	I ₂ C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB 总线电压检测
23	SPI0 发送数据 (TX)	UART1 RTS	I ₂ C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB 总线使能 (VBUS EN)
24	SPI1 RX	UART1 TX	I ₂ C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB 过流检测
25	SPI1 CSn	UART1 RX	I ₂ C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB 总线电压检测
26	SPI1 SCK	UART1 CTS	I ₂ C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
27	SPI1 TX	UART1 RTS	I ₂ C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 过流检测
28	SPI1 RX	UART0 TX	I ₂ C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 总线电压检测
29	SPI1 CSn	UART0 接收 (RX)	I ₂ C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)

每个GPIO一次仅能选择一个功能。同样，每个外设输入（例如UART0 RX）一次应仅选择一个GPIO。若同一外设输入连接至多个GPIO，该外设将接收到这些GPIO输入的逻辑“或”值。

表200。 GPIO用户组功能
描述

功能名称	描述
SPIx	将内部 PL022 SPI 外设之一连接至 GPIO
UARTx	将内部 PL011 UART 外设之一连接至 GPIO
I ₂ Cx	将内部 DW I ₂ C 外设之一连接至 GPIO
PWMx A/B	将PWM切片连接至GPIO。共有八个PWM切片，每个切片包含两个输出通道（A/B）。B引脚亦可用作输入，用于频率和占空比的测量。
SIO	通过单周期IO (SIO) 模块实现GPIO的软件控制。必须选择SIO功能 (F5) 以使处理器驱动GPIO，但输入端始终连接，因此软件能够随时检测GPIO状态。
PIOx	将其中一个可编程IO块 (PIO) 连接至GPIO。PIO可实现多种接口，并具备自身的内部引脚映射硬件，允许用户组GPIO上灵活布置数字接口。必须选择PIO功能 (F6, F7) 以使PIO驱动GPIO，但输入始终连接，因此PIO始终能够读取所有引脚状态。
CLOCK GPINx	通用时钟输入。可路由至RP2040的多个内部时钟域，例如为RTC提供1Hz时钟，或连接至内部频率计数器。
CLOCK GPOUTx	通用时钟输出。可驱动多个内部时钟至GPIO，并支持可选的整数分频。
USB OVCUR DET/VBUS DET/VBUS EN	内部USB控制器的USB电源控制信号。

表201。通用输入/输出 (GPIO) QSPI 银行功能

功能	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9
QSPI SCK	XIP SCK					SIO				
QSPI CSn	XIP CSn					SIO				
QSPI SD0	XIP SD0					SIO				

	功能									
QSPI SD1	XIP SD1					SIO				
QSPI SD2	XIP SD2					SIO				
QSPI SD3	XIP SD3					SIO				

表 282 GPIO QSPI
银行功能
说明

功能名称	描述
XIP	连接至闪存就地执行（XIP）子系统内的同步串行接口（SSI）。此功能使处理器能够直接从外部 SPI、双 SPI 或四线 SPI 闪存中执行代码。
SIO	通过单周期IO（SIO）模块实现GPIO的软件控制。必须选择SIO功能（F5）以驱动GPIO，但输入始终连接，因此软件可随时检测GPIO状态。QSPI IO通过SIO_GPIO_HI_x寄存器控制，且按SCK、CSn、SD0、SD1、SD2、SD3的顺序映射到寄存器位，从最低有效位开始。

这六个QSPI Bank GPIO引脚通常由XIP外设用于与外部闪存设备通信。

然而，存在两种情况下，这些引脚可用作软件控制的GPIO：

- 若采用SPI或Dual-SPI闪存设备执行就地（execute-in-place），则SD2和SD3引脚不用于闪存访问，可用于电路板上的其他GPIO功能。
- 若RP2040采用无闪存配置（仅支持USB启动），则所有六个引脚均可用作软件控制的GPIO功能。

2.19.3. 中断

每个GPIO引脚在以下四种情形下均可触发中断：

- 高电平：GPIO引脚处于逻辑1状态
- 低电平：GPIO引脚处于逻辑0状态
- 上升沿：GPIO从逻辑0跳变至逻辑1
- 下降沿：GPIO从逻辑1跳变至逻辑0

电平中断不会被锁存。这意味着，当引脚为逻辑1且高电平中断处于激活状态时，一旦引脚变为逻辑0，该中断将立即失效。边缘中断的信息存储于 INTR 寄存器，可通过向 INTR 寄存器写入数据予以清除。

针对三类中断目标：proc 0、proc 1 和 dormant_wake，分别设有使能、状态及强制寄存器。针对 proc 0，相关寄存器分别为使能（PROC0_INTE0）、状态（PROC0_INTS0）及强制（PROC0_INTFO）。休眠唤醒用于将 ROSC 或 XOSC 从休眠模式中唤醒。有关休眠模式的详细信息，请参见第2.11.5.2节。

所有中断均在每个总线及每个目标处进行逻辑或运算，合计生成六个GPIO中断：

- IO总线0至休眠唤醒
- IO总线0至处理器0
- IO总线0至处理器1
- IO QSPI至休眠唤醒
- IO QSPI至处理器0
- IO QSPI至处理器1

这意味着用户可以同时监控多个GPIO事件。

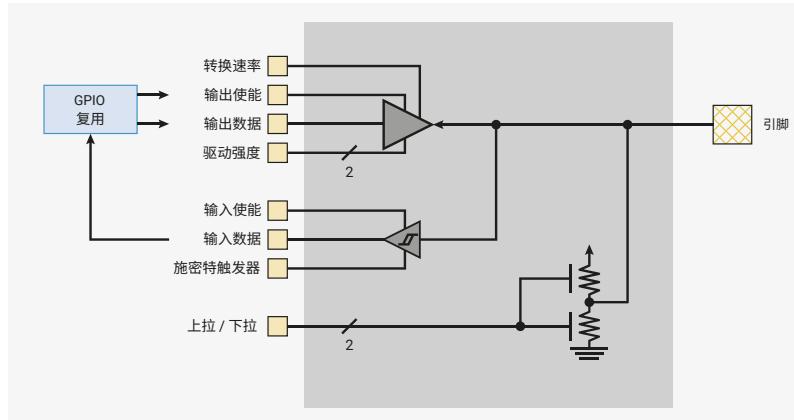
2.19.4. 引脚

每个GPIO通过“引脚”连接到芯片外部世界。引脚是芯片内部逻辑与外部电路之间的电气接口。引脚转换信号电压水平，支持更高电流，并为静电放电（ESD）事件提供一定的防护。引脚的电气特性可调节，以满足外部电路的需求。提供以下调整选项：

- 输出驱动强度可设置为 2mA、4mA、8mA 或 12mA
- 输出转换速率可设置为慢速或快速
- 可启用输入迟滞（施密特触发器模式）
- 可启用上拉或下拉，以在输出驱动器禁用时设定输出信号电平
- 输入缓冲区可被禁用，以降低在引脚未使用、未连接或连接至模拟信号时的电流消耗。

图 37 展示了一个示例引脚。

图 37。单个 IO 引脚的示意图



该引脚的输出使能、输出数据和输入数据端口通过 IO 复用连接至控制该引脚功能。所有其他端口由引脚控制寄存器管理。该寄存器还允许通过覆盖控制该引脚功能的输出使能信号来禁用该引脚的输出驱动器。有关引脚控制寄存器的示例，请参见 GPIO0。

输出信号电平及垫片的可接受输入信号电平均由数字IO电源（IOVDD）决定。

IOVDD可为1.8V至3.3V之间的任意标称电压，但为满足1.8V供电时的规范，必须通过向垫片VOLTAGE_SELECT寄存器写入1以调整垫片输入阈值。默认情况下，垫片输入阈值适用于IOVDD电压在2.5V至3.3V之间。使用1.8V电压且保持默认输入阈值为安全操作模式，但输入阈值将不符合规格要求。

警告

当输入阈值设为1.8V时，使用高于1.8V的IOVDD电压可能会对芯片造成损坏。

垫片输入阈值按银行分别调整，对用户IO银行（IO Bank 0）和QSPI IO银行的垫片分别设有各自的VOLTAGE_SELECT寄存器。然而，两个寄存器组共享相同的数字IO电源（IOVDD），因此两个寄存器应始终设置为相同的值。

寄存器详细信息详见第2.19.6.3节“引脚控制 - 用户寄存器组”和第2.19.6.4节“引脚控制 - QSPI 寄存器组”。

2.19.4.1. 总线保持模式

对于每个引脚，在任何时刻只能启用上拉电阻或下拉电阻其中之一。不可能同时启用两者。相反，若同时设置GPIO0.PDE 和GPIO0.PUE位，则启用总线保持

模式，在该模式下引脚：

- 当输入为高电平时被上拉，
- 当输入为低电平时被下拉。

当输出缓冲器被禁用且引脚未被任何外部信号驱动时，此模式能够弱保持引脚当前的逻辑状态。引脚不会漂浮至中间电平。

2.19.5. 软件示例

2.19.5.1. 选择一个IO功能

一个IO引脚可以执行多种不同功能，必须在使用前进行配置。例如，您可能希望它作为 `UART_TX` 引脚，或作为 `PWM` 输出。SDK提供了`gpio_set_function`函数以实现此目的。许多SDK示例会在初始化时调用`gpio_set_function`，以便通过UART进行打印。

SDK首先定义了一个结构体，用于表示IO银行0，即用户IO银行的寄存器。每个IO均包含一个状态寄存器，紧接其后的是一个控制寄存器。共有30个IO，因此包含状态和控制寄存器的结构体实例被定义为 `io[30]`，实现了30次重复。

SDK链接：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/io_bank0.h 第181行至第229行

```

181 typedef struct {
182     io_bank0_status_ctrl_hw_t io[30];
183
184 // (描述摘自数组索引0寄存器IO_BANK0_INTR0，其他数组索引适用同样规则)
185     _REG_(IO_BANK0_INTR0_OFFSET) // IO_BANK0_INTR0
186     // 原始中断
187     // 0x80000000 [31]      GPIO7_EDGE_HIGH (0)
188     // 0x40000000 [30]      GPIO7_EDGE_LOW (0)
189     // 0x20000000 [29]      GPIO7_LEVEL_HIGH (0)
190     // 0x10000000 [28]      GPIO7_LEVEL_LOW (0)
191     // 0x08000000 [27]      GPIO6_EDGE_HIGH (0)
192     // 0x04000000 [26]      GPIO6_EDGE_LOW (0)
193     // 0x02000000 [25]      GPIO6_LEVEL_HIGH (0)
194     // 0x01000000 [24]      GPIO6_LEVEL_LOW (0)
195     // 0x00800000 [23]      GPIO5_EDGE_HIGH (0)
196     // 0x00400000 [22]      GPIO5_EDGE_LOW (0)
197     // 0x00200000 [21]      GPIO5_LEVEL_HIGH (0)
198     // 0x00100000 [20]      GPIO5_LEVEL_LOW (0)
199     // 0x00080000 [19]      GPIO4_EDGE_HIGH (0)
200     // 0x00040000 [18]      GPIO4_EDGE_LOW (0)
201     // 0x00020000 [17]      GPIO4_LEVEL_HIGH (0)
202     // 0x00010000 [16]      GPIO4_LEVEL_LOW (0)
203     // 0x00008000 [15]      GPIO3_EDGE_HIGH (0)
204     // 0x00004000 [14]      GPIO3_EDGE_LOW (0)
205     // 0x00002000 [13]      GPIO3_LEVEL_HIGH (0)
206     // 0x00001000 [12]      GPIO3_LEVEL_LOW (0)
207     // 0x00000800 [11]      GPIO2_EDGE_HIGH (0)
208     // 0x00000400 [10]      GPIO2_EDGE_LOW (0)
209     // 0x00000200 [9]       GPIO2_LEVEL_HIGH (0)
210     // 0x00000100 [8]       GPIO2_LEVEL_LOW (0)
211     // 0x00000080 [7]       GPIO1_EDGE_HIGH (0)
212     // 0x00000040 [6]       GPIO1_EDGE_LOW (0)
213     // 0x00000020 [5]       GPIO1_LEVEL_HIGH (0)
214     // 0x00000010 [4]       GPIO1_LEVEL_LOW (0)
215     // 0x00000008 [3]       GPIO0_EDGE_HIGH (0)
216     // 0x00000004 [2]       GPIO0_EDGE_LOW (0)

```

```

217     // 0x00000002 [1]      GPIO00_LEVEL_HIGH (0)
218     // 0x00000001 [0]      GPIO00_LEVEL_LOW (0)
219     io_rw_32_intr[4];
220
221     union {
222         struct {
223             io_bank0_irq_ctrl_hw_t proc0_irq_ctrl;
224             io_bank0_irq_ctrl_hw_t proc1_irq_ctrl;
225             io_bank0_irq_ctrl_hw_t dormant_wake_irq_ctrl;
226         };
227         io_bank0_irq_ctrl_hw_t irq_ctrl[3];
228     };
229 } io_bank0_hw_t;

```

为 IO bank 1 的引脚控制寄存器定义了类似结构。默认情况下，所有引脚解除复位后均可直接使用，输入启用，输出禁用设为 0。无论如何，SDK 中的 `gpio_set_function` 函数会设置这些寄存器，以确保引脚可用于所选功能。最后，将所需功能选择写入 IO 控制寄存器（参见 `GPIO0_CTRL` 以获取 IO 控制寄存器示例）。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 36 至 53 行

```

36 // 为此 GPIO 选择功能，并确保引脚输入输出已启用。
37 // 此操作还会清除输入、输出及中断覆盖位。
38 void gpio_set_function(uint gpio, gpio_function_t fn) {
39     check_gpio_param(gpio);
40     invalid_params_if(HARDWARE_GPIO, ((uint32_t)fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB) &
41         ~IO_BANK0_GPIO0_CTRL_FUNCSEL_BITS);
42     // 设置输入使能为开启，输出使能为关闭
43     hw_write_masked(&pads_bank0_hw->io[gpio],
44                     PADS_BANK0_GPIO0_IE_BITS,
45                     PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
46     );
47     // 将除 fsel 以外的所有字段清零；我们希望该 IO 按照外设指示执行。
48     // 这不会影响例如上拉/下拉，因为这些设置位于引脚控制寄存器中。
49     io_bank0_hw->io[gpio].ctrl = fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
50 }

```

2.19.5.2. 启用 GPIO 中断

SDK 提供了当 GPIO 引脚状态变化时触发中断的方法：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 186-196 行

```

186 void gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled) {
187     // 此调用要么禁用中断，要么回调函数应已设置。
188     // 该机制防止在未设置回调函数时启用中断。
189     assert(!enabled || irq_has_handler(IO_IRQ_BANK0));
190
191     // 每个核心维护独立的掩码/强制/状态，因此需检查调用的是哪个核心，
192     // 并设置对应的IRQ控制。
193     io_bank0_irq_ctrl_hw_t *irq_ctrl_base = get_core_num() ?
194         &io_bank0_hw->proc1_irq_ctrl : &io_bank0_hw-
195         >proc0_irq_ctrl;
196     _gpio_set_irq_enabled(gpio, events, enabled, irq_ctrl_base);
197 }

```

`gpio_set_irq_enabled` 调用了更底层函数 `_gpio_set_irq_enabled`：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 173 至 184 行

```

173 static void _gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled,
174     io_bank0_irq_ctrl_hw_t *irq_ctrl_base) {
175     // 清除可能导致立即虚假中断处理程序执行的陈旧事件
176     gpio_acknowledge_irq(gpio, events);
177 
178     io_rw_32 *en_reg = &irq_ctrl_base->inte[	gpio / 8];
179     events <= 4 * (gpio % 8);
180 
181     if (enabled)
182         hw_set_bits(en_reg, events);
183     else
184         hw_clear_bits(en_reg, events);
185 }
```

用户应提供指向回调函数的指针，当 GPIO 事件发生时将调用该函数。使用该系统的示例应用为 hello_gpio_irq：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_gpio_irq/hello_gpio_irq.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX 许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8 #include "pico/stl.h"
9 #include "hardware/gpio.h"
10
11 #define GPIO_WATCH_PIN 2
12
13 static char event_str[128];
14
15 void gpio_event_string(char *buf, uint32_t events);
16
17 void gpio_callback(uint gpio, uint32_t events) {
18     // 将刚发生的 GPIO 事件存入 event_str
19     // 以便打印
20     gpio_event_string(event_str, events);
21     printf("GPIO %d %s\n", gpio, event_str);
22 }
23
24 int main() {
25     stdio_init_all();
26
27     printf("Hello GPIO IRQ\n");
28     gpio_init(GPIO_WATCH_PIN);
29     gpio_set_irq_enabled_with_callback(GPIO_WATCH_PIN, GPIO_IRQ_EDGE_RISE |
30         GPIO_IRQ_EDGE_FALL, true, &gpio_callback);
31
32     // 永久等待
33     while (1);
34
35
36 static const char *gpio_irq_str[] = {
37     "LEVEL_LOW", // 0x1
38     "LEVEL_HIGH", // 0x2
39     "EDGE_FALL", // 0x4
40     "EDGE_RISE" // 0x8
41 }
```

```

41 };
42
43 void gpio_event_string(char *buf, uint32_t events) {
44     for (uint i = 0; i < 4; i++) {
45         uint mask = (1 << i);
46         if (events & mask) {
47             // 将该事件字符串复制到用户缓冲区
48             const char *event_str = gpio_irq_str[i];
49             while (*event_str != '\0') {
50                 *buf++ = *event_str++;
51             }
52             events &= ~mask;
53
54             // 若有更多事件，则添加“，”
55             if (events) {
56                 *buf++ = ',';
57                 *buf++ = ' ';
58             }
59         }
60     }
61     *buf++ = '\0';
62 }

```

2.19.6. 寄存器列表

2.19.6.1. IO - 用户寄存器组

用户寄存器组的IO寄存器起始地址为 `0x40014000`（在SDK中定义为 `IO_BANK0_BASE`）。

表 283. `IO_BANK0` 寄存器列表

偏移量	名称	说明
0x000	<code>GPIO0_STATUS</code>	GPIO 状态
0x004	<code>GPIO0_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x008	<code>GPIO1_STATUS</code>	GPIO 状态
0x00c	<code>GPIO1_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x010	<code>GPIO2_STATUS</code>	GPIO 状态
0x014	<code>GPIO2_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x018	<code>GPIO3_STATUS</code>	GPIO 状态
0x01c	<code>GPIO3_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x020	<code>GPIO4_STATUS</code>	GPIO 状态
0x024	<code>GPIO4_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x028	<code>GPIO5_STATUS</code>	GPIO 状态
0x02c	<code>GPIO5_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x030	<code>GPIO6_STATUS</code>	GPIO 状态
0x034	<code>GPIO6_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x038	<code>GPIO7_STATUS</code>	GPIO 状态
0x03c	<code>GPIO7_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x040	<code>GPIO8_STATUS</code>	GPIO 状态

偏移量	名称	说明
0x044	GPIO8_CTRL	GPIO 控制，包括功能选择和覆盖。
0x048	GPIO9_STATUS	GPIO 状态
0x04c	GPIO9_CTRL	GPIO 控制，包括功能选择和覆盖。
0x050	GPIO10_STATUS	GPIO 状态
0x054	GPIO10_CTRL	GPIO 控制，包括功能选择和覆盖。
0x058	GPIO11_STATUS	GPIO 状态
0x05c	GPIO11_CTRL	GPIO 控制，包括功能选择和覆盖。
0x060	GPIO12_STATUS	GPIO 状态
0x064	GPIO12_CTRL	GPIO 控制，包括功能选择和覆盖。
0x068	GPIO13_STATUS	GPIO 状态
0x06c	GPIO13_CTRL	GPIO 控制，包括功能选择和覆盖。
0x070	GPIO14_STATUS	GPIO 状态
0x074	GPIO14_CTRL	GPIO 控制，包括功能选择和覆盖。
0x078	GPIO15_STATUS	GPIO 状态
0x07c	GPIO15_CTRL	GPIO 控制，包括功能选择和覆盖。
0x080	GPIO16_STATUS	GPIO 状态
0x084	GPIO16_CTRL	GPIO 控制，包括功能选择和覆盖。
0x088	GPIO17_STATUS	GPIO 状态
0x08c	GPIO17_CTRL	GPIO 控制，包括功能选择和覆盖。
0x090	GPIO18_STATUS	GPIO 状态
0x094	GPIO18_CTRL	GPIO 控制，包括功能选择和覆盖。
0x098	GPIO19_STATUS	GPIO 状态
0x09c	GPIO19_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0a0	GPIO20_STATUS	GPIO 状态
0x0a4	GPIO20_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0a8	GPIO21_STATUS	GPIO 状态
0x0ac	GPIO21_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0b0	GPIO22_STATUS	GPIO 状态
0x0b4	GPIO22_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0b8	GPIO23_STATUS	GPIO 状态
0x0bc	GPIO23_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0c0	GPIO24_STATUS	GPIO 状态
0x0c4	GPIO24_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0c8	GPIO25_STATUS	GPIO 状态
0x0cc	GPIO25_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0d0	GPIO26_STATUS	GPIO 状态

偏移量	名称	说明
0x0d4	GPIO26_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0d8	GPIO27_STATUS	GPIO 状态
0x0dc	GPIO27_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0e0	GPIO28_STATUS	GPIO 状态
0x0e4	GPIO28_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0e8	GPIO29_STATUS	GPIO 状态
0x0ec	GPIO29_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0f0	中断 0	原始中断
0x0f4	中断 1	原始中断
0x0f8	中断 2	原始中断
0x0fc	中断 3	原始中断
0x100	PROC0_INTE0	proc0 的中断使能
0x104	PROC0_INTE1	proc0 的中断使能
0x108	PROC0_INTE2	proc0 的中断使能
0x10c	PROC0_INTE3	proc0 的中断使能
0x110	PROC0_INTFO	proc0 的中断强制
0x114	PROC0_INTF1	proc0 的中断强制
0x118	PROC0_INTF2	proc0 的中断强制
0x11c	PROC0_INTF3	proc0 的中断强制
0x120	PROC0_INTS0	proc0 的中断状态 (掩码及强制后)
0x124	PROC0_INTS1	proc0 的中断状态 (掩码及强制后)
0x128	PROC0_INTS2	proc0 的中断状态 (掩码及强制后)
0x12c	PROC0_INTS3	proc0 的中断状态 (掩码及强制后)
0x130	PROC1_INTE0	proc1 的中断使能
0x134	PROC1_INTE1	proc1 的中断使能
0x138	PROC1_INTE2	proc1 的中断使能
0x13c	PROC1_INTE3	proc1 的中断使能
0x140	PROC1_INTFO	proc1 的中断强制
0x144	PROC1_INTF1	proc1 的中断强制
0x148	PROC1_INTF2	proc1 的中断强制
0x14c	PROC1_INTF3	proc1 的中断强制
0x150	PROC1_INTS0	proc1 的中断状态 (掩码及强制后)
0x154	PROC1_INTS1	proc1 的中断状态 (掩码及强制后)
0x158	PROC1_INTS2	proc1 的中断状态 (掩码及强制后)
0x15c	PROC1_INTS3	proc1 的中断状态 (掩码及强制后)
0x160	DORMANT_WAKE_INTE0	dormant_wake 的中断使能

偏移量	名称	说明
0x164	DORMANT_WAKE_INTE1	dormant_wake 的中断使能
0x168	DORMANT_WAKE_INTE2	dormant_wake 的中断使能
0x16c	DORMANT_WAKE_INTE3	dormant_wake 的中断使能
0x170	DORMANT_WAKE_INTFO	dormant_wake 的中断强制
0x174	DORMANT_WAKE_INTF1	dormant_wake 的中断强制
0x178	DORMANT_WAKE_INTF2	dormant_wake 的中断强制
0x17c	DORMANT_WAKE_INTF3	dormant_wake 的中断强制
0x180	DORMANT_WAKE_INTS0	dormant_wake 遮蔽及强制后的中断状态
0x184	DORMANT_WAKE_INTS1	dormant_wake 遮蔽及强制后的中断状态
0x188	DORMANT_WAKE_INTS2	dormant_wake 遮蔽及强制后的中断状态
0x18c	DORMANT_WAKE_INTS3	dormant_wake 遮蔽及强制后的中断状态

IO_BANK0: GPIO0_STATUS, GPIO1_STATUS, ..., GPIO28_STATUS, GPIO29_STATUS 寄存器

偏移量: 0x000, 0x008, ..., 0x0e0, 0x0e8

描述

GPIO 状态

表 284.
GPIO0_STATUS,
GPIO1_STATUS, ...,
GPIO28_STATUS,
GPIO29_STATUS
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	IRQTOPROC : 应用覆盖后传递给处理器的中断	只读	0x0
25	保留。	-	-
24	IRQFROMPAD : 应用覆盖前来自引脚的中断	只读	0x0
23:20	保留。	-	-
19	INTOPERI : 应用覆盖后传入外设的信号	只读	0x0
18	保留。	-	-
17	INFROMPAD : 应用覆盖前来自引脚的输入信号	只读	0x0
16:14	保留。	-	-
13	OETOPAD : 应用寄存器覆盖后传给引脚的输出使能	只读	0x0
12	OEFROMPERI : 应用寄存器覆盖前来自选定外设的输出使能	只读	0x0
11:10	保留。	-	-
9	OUTTOPAD : 应用寄存器覆盖后传给引脚的输出信号	只读	0x0
8	OUTFROMPERI : 所选外设的输出信号，在寄存器覆盖生效前	只读	0x0
7:0	保留。	-	-

IO_BANK0: GPIO0_CTRL、GPIO1_CTRL、...、GPIO28_CTRL、GPIO29_CTRL

寄存器

偏移量：0x004、0x00c、...、0x0e4、0x0ec

描述

GPIO 控制，包括功能选择和覆盖。

表285。
GPIO0_CTRL,
GPIO1_CTRL, ...
GPIO28_CTRL,
GPIO29_CTRL
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:28	IRQOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：中断不反转		
	0x1 → INVERT：中断反转		
	0x2 → LOW：中断拉低		
	0x3 → HIGH：中断拉高		
27:18	保留。	-	-
17:16	INOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：外设输入不反转		
	0x1 → INVERT：外设输入反转		
	0x2 → LOW：外设输入拉低		
	0x3 → HIGH：外设输入拉高		
15:14	保留。	-	-
13:12	OEOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出使能，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出使能，由funcsel选择的外设信号的反相信号控制		
	0x2 → DISABLE：禁用输出		
	0x3 → ENABLE：使能输出		
11:10	保留。	-	-
9:8	OUTOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出，由funcsel选择的外设信号的反相信号控制		
	0x2 → LOW：驱动输出低电平		
	0x3 → HIGH：驱动输出高电平		
7:5	保留。	-	-

位	描述	类型	复位值
4:0	FUNCSEL : 功能选择。31表示NULL。具体可用功能请参见GPIO功能表。	读写	0x1f

IO_BANK0: INTRO寄存器

偏移: 0x0f0

描述

原始中断

表 286. INTRO
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	WC	0x0
30	GPIO7_EDGE_LOW	WC	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	WC	0x0
26	GPIO6_EDGE_LOW	WC	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	WC	0x0
22	GPIO5_EDGE_LOW	WC	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	WC	0x0
18	GPIO4_EDGE_LOW	WC	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	WC	0x0
14	GPIO3_EDGE_LOW	WC	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	WC	0x0
10	GPIO2_EDGE_LOW	WC	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	WC	0x0
6	GPIO1_EDGE_LOW	WC	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
2	GPIO0_EDGE_LOW	WC	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0：INTR1寄存器

偏移量：0x0f4

描述

原始中断

表 287. INTR1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	WC	0x0
30	GPIO15_EDGE_LOW	WC	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	WC	0x0
26	GPIO14_EDGE_LOW	WC	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	WC	0x0
22	GPIO13_EDGE_LOW	WC	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	WC	0x0
18	GPIO12_EDGE_LOW	WC	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	WC	0x0
14	GPIO11_EDGE_LOW	WC	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	WC	0x0
10	GPIO10_EDGE_LOW	WC	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	WC	0x0
6	GPIO9_EDGE_LOW	WC	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0

位	描述	类型	复位值
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	WC	0x0
2	GPIO8_EDGE_LOW	WC	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: INTR2 寄存器

偏移: 0x0f8

描述

原始中断

表 288. INTR2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	WC	0x0
30	GPIO23_EDGE_LOW	WC	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	WC	0x0
26	GPIO22_EDGE_LOW	WC	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	WC	0x0
22	GPIO21_EDGE_LOW	WC	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	WC	0x0
18	GPIO20_EDGE_LOW	WC	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	WC	0x0
14	GPIO19_EDGE_LOW	WC	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	WC	0x0
10	GPIO18_EDGE_LOW	WC	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
6	GPIO17_EDGE_LOW	WC	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	WC	0x0
2	GPIO16_EDGE_LOW	WC	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: INTR3 寄存器

偏移: 0x0fc

描述

原始中断

表 289. INTR3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	WC	0x0
22	GPIO29_EDGE_LOW	WC	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	WC	0x0
18	GPIO28_EDGE_LOW	WC	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	WC	0x0
14	GPIO27_EDGE_LOW	WC	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	WC	0x0
10	GPIO26_EDGE_LOW	WC	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	WC	0x0
6	GPIO25_EDGE_LOW	WC	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	WC	0x0
2	GPIO24_EDGE_LOW	WC	0x0

位	描述	类型	复位值
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTE0 寄存器

偏移: 0x100

描述

proc0 的中断使能

表 290.
PROC0_INTE0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTE1 寄存器

偏移: 0x104

描述

proc0 的中断使能

表 291.
PROC0_INTE1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0

位	描述	类型	复位值
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0：PROC0_INTE2 寄存器

偏移: 0x108

描述

proc0 的中断使能

表 292.
PROC0_INTE2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0：PROC0_INTE3 寄存器

偏移: 0x10c

描述

proc0 的中断使能

表293。
PROC0_INTE3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTFO 寄存器

偏移量: 0x110

描述

proc0 的中断强制

表294。
PROC0_INTFO 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF1 寄存器

偏移量: 0x114

描述

proc0 的中断强制

表295。
PROC0_INTF1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF2 寄存器

偏移量: 0x118

描述

proc0 的中断强制

表296。
PROC0_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF3 寄存器

偏移量: 0x11c

描述

proc0 的中断强制

表297。
PROC0_INTF3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTS0 寄存器

偏移: 0x120

说明

proc0 的中断状态（掩码及强制后）

表 298
PROC0_INTS0
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0

位	描述	类型	复位值
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTS1 寄存器

偏移: 0x124

说明

proc0 的中断状态 (掩码及强制后)

表 299
PROC0_INTS1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0：PROC0_INTS2 寄存器

偏移: 0x128

说明

proc0 的中断状态（掩码及强制后）

表 300
PROC0_INTS2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0

位	描述	类型	复位值
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTS3 寄存器

偏移: 0x12c

说明

proc0 的中断状态 (掩码及强制后)

表 301
PROC0_INTS3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0

位	描述	类型	复位值
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTE0 寄存器

偏移: 0x130

说明

proc1 的中断使能

表 302。
PROC1_INTE0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE1 寄存器

偏移量: 0x134

说明

proc1 的中断使能

表 303。
PROC1_INTE1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE2 寄存器

偏移量: 0x138

说明

proc1 的中断使能

表 304。
PROC1_INTE2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE3 寄存器

偏移量: 0x13c

说明

proc1 的中断使能

表 305。
PROC1_INTE3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0

位	描述	类型	复位值
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTO 寄存器

偏移量: 0x140

描述

proc1 的中断强制

表 306。
PROC1_INTO 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0：PROC1_INTF1 寄存器

偏移：0x144

描述

proc1 的中断强制

表 307
PROC1_INTF1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0

位	描述	类型	复位值
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTF2 寄存器

偏移: 0x148

描述

proc1 的中断强制

表 308
PROC1_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0：PROC1_INTF3 寄存器

偏移：0x14c

描述

proc1 的中断强制

表 309
PROC1_INTF3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTS0 寄存器

偏移: 0x150

描述

proc1 的中断状态（掩码及强制后）

表 310
PROC1_INTS0
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0

位	描述	类型	复位值
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS1 寄存器

偏移: 0x154

描述

proc1 的中断状态（掩码及强制后）

表311
PROC1_INTS1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0

位	描述	类型	复位值
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS2寄存器

偏移量: 0x158

说明

proc1 的中断状态 (掩码及强制后)

表312
PROC1_INTS2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0

位	描述	类型	复位值
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS3寄存器

偏移量: 0x15c

说明

proc1 的中断状态 (掩码及强制后)

表313
PROC1_INTS3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTE0寄存器

偏移量: 0x160

说明

dormant_wake 的中断使能

表314
DORMANT_WAKE_INT
E0寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0

位	描述	类型	复位值
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE1寄存器

偏移量: 0x164

说明

dormant_wake 的中断使能

表315
DORMANT_WAKE_INT
E1寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0

位	描述	类型	复位值
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE2 寄存器

偏移量: 0x168

说明

dormant_wake 的中断使能

表316。
DORMANT_WAKE_INT
E2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0

位	描述	类型	复位值
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE3 寄存器

偏移量: 0x16c

说明

dormant_wake 的中断使能

表317。
DORMANT_WAKE_INT
E3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INT0 寄存器

偏移量：0x170

说明

dormant_wake 的中断强制

表318。
DORMANT_WAKE_INT
F0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF1 寄存器

偏移量: 0x174

说明

dormant_wake 的中断强制

表319。
DORMANT_WAKE_INT
F1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF2 寄存器

偏移量: 0x178

说明

dormant_wake 的中断强制

表 320。
DORMANT_WAKE_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF3 寄存器

偏移量: 0x17c

描述

dormant_wake 的中断强制

表 321。
DORMANT_WAKE_INT
F3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0

位	描述	类型	复位值
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTS0 寄存器

偏移量: 0x180

描述

dormant_wake 遮蔽及强制后的中断状态

表 322。
DORMANT_WAKE_INTS0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS1 寄存器

偏移量: 0x184

描述

dormant_wake 遮蔽及强制后的中断状态

表 323。
DORMANT_WAKE_INT
S1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0

位	描述	类型	复位值
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS2 寄存器

偏移量: 0x188

描述

dormant_wake 遮蔽及强制后的中断状态

表 324。
DORMANT_WAKE_INT
S2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS3 寄存器

偏移量: 0x18c

描述

dormant_wake 遮蔽及强制后的中断状态

表 325。
DORMANT_WAKE_INT
S3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0

位	描述	类型	复位值
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

2.19.6.2. IO - QSPI 银行

QSPI 银行 IO 寄存器起始基地址为 **0x40018000** (在 SDK 中定义为 **IO_QSPI_BASE**)。

表 326。IO_QSPI
寄存器列表

偏移量	名称	说明
0x00	GPIO_QSPI_SCLK_STATUS	GPIO 状态
0x04	GPIO_QSPI_SCLK_CTRL	GPIO 控制，包括功能选择和覆盖。
0x08	GPIO_QSPI_SS_STATUS	GPIO 状态
0x0c	GPIO_QSPI_SS_CTRL	GPIO 控制，包括功能选择和覆盖。
0x10	GPIO_QSPI_SD0_STATUS	GPIO 状态
0x14	GPIO_QSPI_SD0_CTRL	GPIO 控制，包括功能选择和覆盖。
0x18	GPIO_QSPI_SD1_STATUS	GPIO 状态
0x1c	GPIO_QSPI_SD1_CTRL	GPIO 控制，包括功能选择和覆盖。
0x20	GPIO_QSPI_SD2_STATUS	GPIO 状态
0x24	GPIO_QSPI_SD2_CTRL	GPIO 控制，包括功能选择和覆盖。
0x28	GPIO_QSPI_SD3_STATUS	GPIO 状态
0x2c	GPIO_QSPI_SD3_CTRL	GPIO 控制，包括功能选择和覆盖。
0x30	INTR	原始中断
0x34	PROC0_INTE	proc0 的中断使能
0x38	PROC0_INTF	proc0 的中断强制
0x3c	PROC0_INTS	proc0 的中断状态 (掩码及强制后)
0x40	PROC1_INTE	proc1 的中断使能
0x44	PROC1_INTF	proc1 的中断强制
0x48	PROC1_INTS	proc1 的中断状态 (掩码及强制后)
0x4c	DORMANT_WAKE_INTE	dormant_wake 的中断使能

偏移量	名称	说明
0x50	DORMANT_WAKE_INTF	dormant_wake 的中断强制
0x54	DORMANT_WAKE_INTS	dormant_wake 遮蔽及强制后的中断状态

IO_QSPI: GPIO_QSPI_SCLK_STATUS, GPIO_QSPI_SS_STATUS, ..., GPIO_QSPI_SD2_STATUS, GPIO_QSPI_SD3_STATUS 寄存器

偏移量: 0x00, 0x08, ..., 0x20, 0x28

描述

GPIO 状态

表 327。
GPIO_QSPI_SCLK_STA
TUS,
GPIO_QSPI_SS_STATU
S, ...
GPIO_QSPI_SD2_STAT
US,
GPIO_QSPI_SD3_STA
TUS 寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	IRQTOPROC : 应用覆盖后传递给处理器的中断	只读	0x0
25	保留。	-	-
24	IRQFROMPAD : 应用覆盖前来自引脚的中断	只读	0x0
23:20	保留。	-	-
19	INTOPERI : 应用覆盖后传入外设的信号	只读	0x0
18	保留。	-	-
17	INFROMPAD : 应用覆盖前来自引脚的输入信号	只读	0x0
16:14	保留。	-	-
13	OETOPAD : 应用寄存器覆盖后传给引脚的输出使能	只读	0x0
12	OEFROMPERI : 应用寄存器覆盖前来自选定外设的输出使能	只读	0x0
11:10	保留。	-	-
9	OUTTOPAD : 应用寄存器覆盖后传给引脚的输出信号	只读	0x0
8	OUTFROMPERI : 所选外设的输出信号，在寄存器覆盖生效前	只读	0x0
7:0	保留。	-	-

IO_QSPI: GPIO_QSPI_SCLK_CTRL, GPIO_QSPI_SS_CTRL, ..., GPIO_QSPI_SD2_CTRL, GPIO_QSPI_SD3_CTRL 寄存器

偏移量: 0x04, 0x0c, ..., 0x24, 0

x2c 说明

GPIO 控制，包括功能选择和覆盖。

表328。
GPIO_QSPI_SCLK_CTR
L,
GPIO_QSPI_SS_CTRL,
...
GPIO_QSPI_SD2_CTRL,
GPIO_QSPI_SD3_CTRL
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:28	IRQOVER	读写	0x0
	枚举值: 0x0 → NORMAL: 中断不反转		

位	描述	类型	复位值
	0x1 → INVERT：中断反转		
	0x2 → LOW：中断拉低		
	0x3 → HIGH：中断拉高		
27:18	保留。	-	-
17:16	INOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：外设输入不反转		
	0x1 → INVERT：外设输入反转		
	0x2 → LOW：外设输入拉低		
	0x3 → HIGH：外设输入拉高		
15:14	保留。	-	-
13:12	OEOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出使能，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出使能，由funcsel选择的外设信号的反相信号控制		
	0x2 → DISABLE：禁用输出		
	0x3 → ENABLE：使能输出		
11:10	保留。	-	-
9:8	OUTOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出，由funcsel选择的外设信号的反相信号控制		
	0x2 → LOW：驱动输出低电平		
	0x3 → HIGH：驱动输出高电平		
7:5	保留。	-	-
4:0	FUNCSEL ：功能选择。31表示NULL。具体可用功能请参见GPIO功能表。	读写	0x1f

IO_QSPI: INTR 寄存器

偏移量: 0x30

说明

原始中断

表 329. INTR 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
22	GPIO_QSPI_SD3_EDGE_LOW	WC	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	WC	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	WC	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	WC	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	WC	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	WC	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	WC	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	WC	0x0
6	GPIO_QSPI_SS_EDGE_LOW	WC	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	WC	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	WC	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: PROC0_INTE 寄存器

偏移: 0x34

描述

proc0 的中断使能

表330。
PROC0_INTE 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0

位	描述	类型	复位值
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: PROCO_INTF 寄存器

偏移: 0x38

描述

proc0 的中断强制

表331。
PROCO_INTF 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC0_INTS 寄存器

偏移：0x3c

描述

proc0 的中断状态（掩码及强制后）

表 332。
PROC0_INTS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: PROC1_INTE 寄存器

偏移: 0x40

描述

proc1 的中断使能

表 333。
PROC1_INTE 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0

位	描述	类型	复位值
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC1_INTF 寄存器

偏移: 0x44

描述

proc1 的中断强制

表 334。
PROC1_INTF 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC1_INTS 寄存器

偏移量: 0x48

描述

proc1 的中断状态（掩码及强制后）

表 335。
PROC1_INTS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: DORMANT_WAKE_INTE 寄存器

偏移量：0x4c

说明

dormant_wake 的中断使能

表 336。
DORMANT_WAKE_INTE
E 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0

位	描述	类型	复位值
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: DORMANT_WAKE_INTF 寄存器

偏移: 0x50

说明

dormant_wake 的中断强制

表 337。
DORMANT_WAKE_INT
F 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0

位	描述	类型	复位值
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: DORMANT_WAKE_INTS 寄存器

偏移: 0x54

说明

dormant_wake 遮蔽及强制后的中断状态

表338。
DORMANT_WAKE_INT
S寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0

位	描述	类型	复位值
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

2.19.6.3. 用户组 Pad 控制

用户组 Pad 控制寄存器的基地址起始于 `0x4001c000`（在 SDK 中定义为 `PADS_BANK0_BASE`）。

表339。 `PADS_BANK0` 寄存器列表

偏移量	名称	说明
0x00	VOLTAGE_SELECT	电压选择。按组控制
0x04	GPIO0	Pad 控制寄存器
0x08	GPIO1	Pad 控制寄存器
0x0c	GPIO2	Pad 控制寄存器
0x10	GPIO3	Pad 控制寄存器
0x14	GPIO4	Pad 控制寄存器
0x18	GPIO5	Pad 控制寄存器
0x1c	GPIO6	Pad 控制寄存器
0x20	GPIO7	Pad 控制寄存器
0x24	GPIO8	Pad 控制寄存器
0x28	GPIO9	Pad 控制寄存器
0x2c	GPIO10	Pad 控制寄存器
0x30	GPIO11	Pad 控制寄存器
0x34	GPIO12	Pad 控制寄存器
0x38	GPIO13	Pad 控制寄存器
0x3c	GPIO14	Pad 控制寄存器
0x40	GPIO15	Pad 控制寄存器
0x44	GPIO16	Pad 控制寄存器
0x48	GPIO17	Pad 控制寄存器

偏移量	名称	说明
0x4c	GPIO18	Pad 控制寄存器
0x50	GPIO19	Pad 控制寄存器
0x54	GPIO20	Pad 控制寄存器
0x58	GPIO21	Pad 控制寄存器
0x5c	GPIO22	Pad 控制寄存器
0x60	GPIO23	Pad 控制寄存器
0x64	GPIO24	Pad 控制寄存器
0x68	GPIO25	Pad 控制寄存器
0x6c	GPIO26	Pad 控制寄存器
0x70	GPIO27	Pad 控制寄存器
0x74	GPIO28	Pad 控制寄存器
0x78	GPIO29	Pad 控制寄存器
0x7c	SWCLK	Pad 控制寄存器
0x80	SWD	Pad 控制寄存器

PADS_BANK0: VOLTAGE_SELECT 寄存器

偏移: 0x00

表340。
VOLTAGE_SELECT
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	电压选择。按组控制	读写	0x0
	枚举值：		
	0x0 → 3V3：设置电压为3.3V (DVDD ≥ 2V5)		
	0x1 → 1V8：设置电压为1.8V (DVDD ≤ 1V8)		

PADS_BANK0: GPIO0、GPIO1、...、GPIO28、GPIO29 寄存器

偏移量: 0x04、0x08、...、0x74、0x78

描述

Pad 控制寄存器

表341。GPIO0,
GPIO1,...,GPIO28,
GPIO29 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD ：输出禁用。优先于外设的输出使能	读写	0x0
6	IE ：输入使能。	读写	0x1
5:4	DRIVE ：驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		

位	描述	类型	复位值
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x1
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_BANK0: SWCLK 寄存器

偏移: 0x7c

描述

Pad 控制寄存器

表 342. SWCLK 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x1
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值:		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x1
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_BANK0: SWD 寄存器

偏移: 0x80

说明

引脚控制寄存器

表 343. SWD 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值:		
	0x0 → 2mA		

位	描述	类型	复位值
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE : 上拉使能	读写	0x1
2	PDE : 下拉使能	读写	0x0
1	SCHMITT : 施密特触发器使能	读写	0x1
0	SLEWFAST : 转换速率控制。1 = 快, 0 = 慢	读写	0x0

2.19.6.4. 引脚控制 - QSPI 组

QSPI 组引脚控制寄存器的起始基地址为 **0x40020000** (在 SDK 中定义为 PADS_QSPI_BASE)。

表 344. PADS_QSPI 寄存器列表

偏移量	名称	说明
0x00	VOLTAGE_SELECT	电压选择。按组控制
0x04	GPIO_QSPI_SCLK	Pad 控制寄存器
0x08	GPIO_QSPI_SD0	Pad 控制寄存器
0x0c	GPIO_QSPI_SD1	Pad 控制寄存器
0x10	GPIO_QSPI_SD2	Pad 控制寄存器
0x14	GPIO_QSPI_SD3	Pad 控制寄存器
0x18	GPIO_QSPI_SS	Pad 控制寄存器

PADS_QSPI: VOLTAGE_SELECT 寄存器

偏移: 0x00

表 345. VOLTAGE_SELECT 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	电压选择。按组控制 枚举值: 0x0 → 3V3: 设置电压为3.3V (DVDD ≥ 2V5) 0x1 → 1V8: 设置电压为1.8V (DVDD ≤ 1V8)	读写	0x0

PADS_QSPI: GPIO_QSPI_SCLK 寄存器

偏移量: 0x04

描述

引脚控制寄存器

表 346. GPIO_QSPI_SCLK 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD : 输出禁用。优先于外设的输出使能	读写	0x0
6	IE : 输入使能。	读写	0x1

位	描述	类型	复位值
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x1
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_QSPI: GPIO_QSPI_SD0, GPIO_QSPI_SD1, GPIO_QSPI_SD2, GPIO_QS PI_SD3 寄存器

偏移量：0x08, 0x0c, 0x10, 0x14

描述

Pad 控制寄存器

表 347。
GPIO_QSPI_SD0
、GPIO_QSPI_SD1
、GPIO_QSPI_SD2
、GPIO_QSPI_SD3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_QSPI: GPIO_QSPI_SS 寄存器

偏移量：0x18

描述

引脚控制寄存器

表 348。
GPIO_QSPI_SS
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x1
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

2.20. 系统信息

2.20.1. 概述

sysinfo 区块包含系统信息。第一个寄存器包含芯片 ID，允许程序员识别软件运行的芯片版本。设备上的第二个寄存器始终读取为 1。

2.20.2. 寄存器列表

sysinfo 寄存器起始基址为 [0x40000000](#)(定义于SDK中的SYSINFO_BASE)。

表349。SYSINF
0 寄存器列表

偏移量	名称	说明
0x00	CHIP_ID	符合JEDEC JEP-106标准的芯片标识符。
0x04	平台	平台寄存器。允许软件识别其运行环境。
0x40	GITREF_RP2040	芯片源代码的Git哈希值。用于识别芯片版本。

SYSINFO：CHIP_ID 寄存器

偏移: 0x00

描述

符合JEDEC JEP-106标准的芯片标识符。

表350。CHIP_ID
寄存器

位	描述	类型	复位值
31:28	修订版本	只读	-
27:12	部件	只读	-

位	描述	类型	复位值
11:0	制造商	只读	-

SYSINFO：PLATFORM寄存器

偏移量: 0x04

说明

平台寄存器。允许软件识别其运行环境。

表351。PLATFORM
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	ASIC	只读	0x0
0	FPGA	只读	0x0

SYSINFO: GITREF_RP2040 寄存器

偏移: 0x40

表 352。
GITREF_RP2040
寄存器

位	描述	类型	复位值
31:0	芯片源代码的Git哈希值。用于识别芯片版本。	只读	-

2.21. Syscfg

2.21.1. 概述

系统配置块控制包括以下杂项芯片设置：

- NMI（不可屏蔽中断）屏蔽，用以选择产生 NMI 的源
- 处理器配置
 - DAP 实例 ID（用于更改 SWD 与核心调试通信时所用的地址）
 - 处理器状态（处理器是否停止，此信息在调试时可能有用）
- 处理器 IO 配置
 - 输入同步器控制（允许绕过输入同步器以减少时钟同步情况下的延迟）
 - 调试控制
 - 提供从芯片内部控制 SWD 接口的能力。这意味着核心0能够调试核心1，可能使调试连接更为便捷。
- 存储器断电（未使用时，单个存储器可断电以节省少量额外功耗）。

2.21.2. 寄存器列表

系统配置寄存器起始基址为 **0x40004000**（在 SDK 中定义为 SYSCFG_BASE）。

表 353. SYSCFG 寄存器列表

偏移量	名称	说明
0x00	PROC0_NMI_MASK	处理器核 0 NMI 源掩码
0x04	PROC1_NMI_MASK	处理器核 1 NMI 源掩码
0x08	PROC_CONFIG	处理器配置
0x0c	PROC_IN_SYNC_BYPASS	对于每个位，若为 1，则绕过该 GPIO 与 SIO 中 GPIO 输入寄存器之间的输入同步器。输入同步器通常应 保持不绕过，以避免向处理器注入亚稳态。 若您有足够的把握，可绕过同步器以节省两个输入 延迟周期。该寄存器适用于 GPIO 0 至 29。
0x10	PROC_IN_SYNC_BYPASS_HI	对于每个位，若为 1，则绕过该 GPIO 与 SIO 中 GPIO 输入寄存器之间的输入同步器。输入同步器通常应 保持不绕过，以避免向处理器注入亚稳态。 若您有足够的把握，可绕过同步器以节省两个输入 延迟周期。该寄存器适用于 GPIO 30 至 35 (QSPI IO)。
0x14	DBGFORCE	直接控制任一处理器的 SWD 调试端口
0x18	MEMPOWERDOWN	控制内存的掉电状态。置高电平以关闭存储器电源。 请务必谨慎使用

SYSCFG: PROC0_NMI_MASK 寄存器

偏移: 0x00

描述

处理器核 0 NMI 源掩码

表 354.
PROC0_NMI_MASK
寄存器

位	描述	类型	复位值
31:0	将对应位设置为高以启用来自该 IRQ 的 NMI	读写	0x00000000

SYSCFG: PROC1_NMI_MASK 寄存器

偏移量: 0x04

说明

处理器核 1 NMI 源掩码

表 355.
PROC1_NMI_MASK
寄存器

位	描述	类型	复位值
31:0	将对应位设置为高以启用来自该 IRQ 的 NMI	读写	0x00000000

SYSCFG: PROC_CONFIG 寄存器

偏移量: 0x08

说明

处理器配置

表 356。
PROC_CONFIG
寄存器

位	描述	类型	复位值
31:28	PROC1_DAP_INSTID : 配置 proc1 DAP 实例 ID。 建议仅在多芯片环境中需要调试访问时更改此项 警告：请勿设置为 15，该值保留供 RescueDP 使用	读写	0x1
27:24	PROC0_DAP_INSTID : 配置 proc0 DAP 实例 ID。 建议仅在多芯片环境中需要调试访问时更改此项 警告：请勿设置为 15，该值保留供 RescueDP 使用	读写	0x0
23:2	保留。	-	-
1	PROC1_HALTED : 指示 proc1 已停止	只读	0x0
0	PROC0_HALTED : 指示 proc0 已停止	只读	0x0

SYSCFG: PROC_IN_SYNC_BYPASS 寄存器

偏移: 0x0c

表 357。
PROC_IN_SYNC_BYPA
SS 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对于每个位，如果为1，则绕过SIO中该GPIO与GPIO输入寄存器之间的输入同步器。一般应避免绕过输入同步器，以防将亚稳态注入处理器。 如果您有胆量，可以通过绕过来节省两个周期的输入延迟。该寄存器适用于 GPIO 0 至 29。	读写	0x00000000

SYSCFG: PROC_IN_SYNC_BYPASS_HI 寄存器

偏移: 0x10

表 358。
PROC_IN_SYNC_BYP
ASS_HI 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对于每个位，如果为1，则绕过SIO中该GPIO与GPIO输入寄存器之间的输入同步器。一般应避免绕过输入同步器，以防将亚稳态注入处理器。 如果您有胆量，可以通过绕过来节省两个周期的输入延迟。该寄存器适用于 GPIO 30 至 35 (QSPI IO)。	读写	0x00

SYSCFG: DBGFORCE 寄存器

偏移: 0x14

说明

直接控制任一处理器的 SWD 调试端口

表 359: DBGFORCE
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	PROC1_ATTACH : 将处理器1调试端口连接至syscfg控制，并断开其与外部SWD引脚的连接。	读写	0x0
6	PROC1_SWCLK : 若设置了PROC1_ATTACH，则直接驱动处理器1的SWCLK信号。	读写	0x1

位	描述	类型	复位值
5	PROC1_SWDI : 若设置了PROC1_ATTACH，则直接驱动处理器1的SWDIO输入信号。	读写	0x1
4	PROC1_SWDO : 观察处理器1的SWDIO输出信号值。	只读	-
3	PROC0_ATTACH : 将处理器0的调试端口连接至syscfg控制单元，同时断开其与外部SWD焊盘的连接。	读写	0x0
2	PROC0_SWCLK : 若设置了PROC0_ATTACH，则直接驱动处理器0的SWCLK信号。	读写	0x1
1	PROC0_SWDI : 若设置了PROC0_ATTACH，则直接驱动处理器0的SWDIO输入信号。	读写	0x1
0	PROC0_SWDO : 监测处理器0的SWDIO输出值。	只读	-

SYSCFG：MEMPOWERDOWN寄存器

偏移：0x18

说明

控制内存的掉电状态。置高以断电存储器。

请务必谨慎使用

表360。
MEMPOWERDOWN
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	ROM	读写	0x0
6	USB	读写	0x0
5	SRAM5	读写	0x0
4	SRAM4	读写	0x0
3	SRAM3	读写	0x0
2	SRAM2	读写	0x0
1	SRAM1	读写	0x0
0	SRAM0	读写	0x0

2.22. TBMAN

TBMAN指测试台管理器，用于芯片开发仿真中验证设计的功能。

在这些仿真过程中，TBMAN允许运行于RP2040上的软件控制测试台及仿真环境。在真实芯片中，其唯一作用是提供一个单一的 PLATFORM 寄存器，以表明芯片为真实硬件。该 PLATFORM 功能在 sysinfo (第2.20节) 寄存器中亦有重复实现。

2.22.1. 寄存器列表

TBMAN 寄存器起始地址为 **0x4006c000** (在 SDK 中定义为 TBMAN_BASE)。

表 361.
TBMAN 寄存器列表

偏移量	名称	说明
0x0	平台	指示所使用的平台类型

TBMAN：PLATFORM 寄存器

偏移: 0x0

描述

指示所使用的平台类型

表 362.
PLATFORM 寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	FPGA : 表示平台为 FPGA	只读	0x0
0	ASIC : 表示平台为 ASIC	只读	0x1

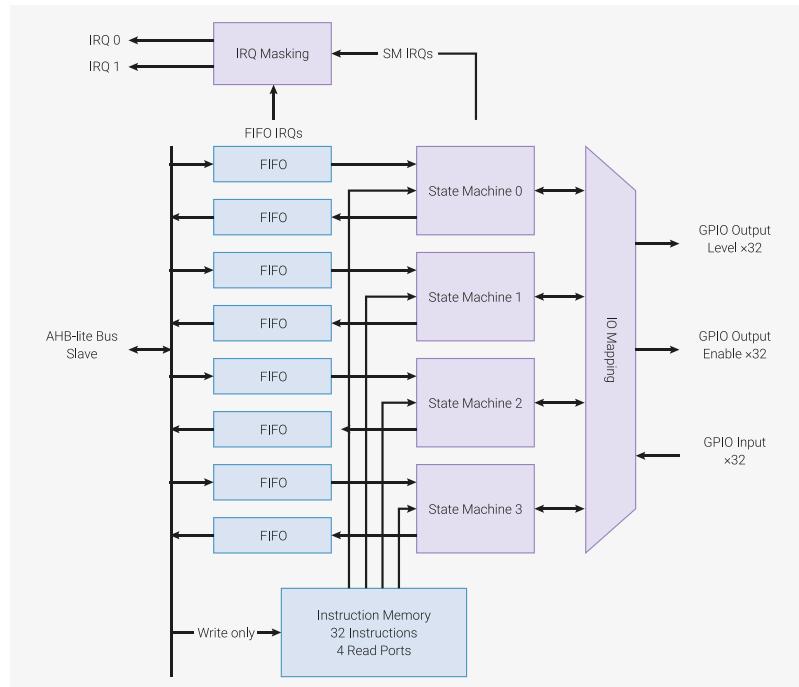
第3章. PIO

3.1. 概述

RP2040 中有两个相同的 PIO 模块。每个 PIO 模块均具备专用的总线结构、GPIO 和中断控制器连接。单个 PIO 模块的示意图如图 38 所示。

图 38。PIO 模块结构示意图。共有两个 PIO 模块，每个模块包含四个状态机。这四个状态机同时从共享的指令存储器执行程序。

FIFO 数据队列缓冲在 PIO 与系统之间传输的数据。GPIO 映射逻辑允许每个状态机监视并操作最多 30 个 G PIO。



可编程输入/输出模块（PIO）是一种多功能硬件接口。它支持多种输入输出标准，包括：

- 8080和6800并行总线
- I2C
- 3针I2S
- SDIO
- SPI、DSPI、QSPI
- UART
- DPI或VGA（通过电阻式数模转换器）

PIO 的编程方式与处理器类似。共有两个 PIO 模块，每个模块包含四个状态机，可独立执行顺序程序以控制 GPIO 及传输数据。与通用处理器不同，PIO 状态机高度专用于 IO，强调确定性、精确时序及与定功能硬件的紧密集成。每个状态机配备：

- 两个32位移位寄存器——任意方向，任意移位位数
- 两个32位临时寄存器
- 4×32位总线FIFO，双向（发送/接收），可重新配置为单向8×32
- 分数时钟分频器（16位整数，8位小数）

- 灵活的GPIO映射
- DMA接口，系统DMA可持续提供高达每时钟周期1个字的吞吐量
- IRQ标志的设置／清除／状态

每个状态机及其支持硬件占用的硅片面积大致相当于标准串行接口模块，例如SPI或I2C控制器。然而，PIO状态机可动态配置和重新配置，以实现多种不同接口。

通过将状态机设计为类似软件的可编程方式，而非像CPLD那样的完全可配置逻辑结构，可在相同成本和功耗范围内提供更多硬件接口。这同时为希望通过直接编程而非使用预置PIO库接口充分利用PIO灵活性的用户，提供了更为熟悉的编程模型和更简便的工具流程。

PIO不仅灵活且高性能，这得益于每个状态机内经过精心挑选的固定功能硬件。在以48MHz系统时钟运行并输出DPI时，PIO能在活动扫描线期间持续提供360Mbps的传输速率。在本示例中，一个状态机负责帧和扫描线时序并生成像素时钟，另一个状态机负责像素数据，并对运行长度编码的扫描线进行解码。

状态机的输入和输出可映射至最多32个GPIO（RP2040限制为30个GPIO），且所有状态机均可独立且同时访问任意GPIO。例如，标准UART代码允许TX、RX、CTS和RTS为任意四个GPIO，I2C同样允许SDA和SCL为任意GPIO。可用自由度取决于特定PIO程序如何利用PIO的引脚映射资源，但至少可以在一定范围内自由调整接口对应的GPIO编号。

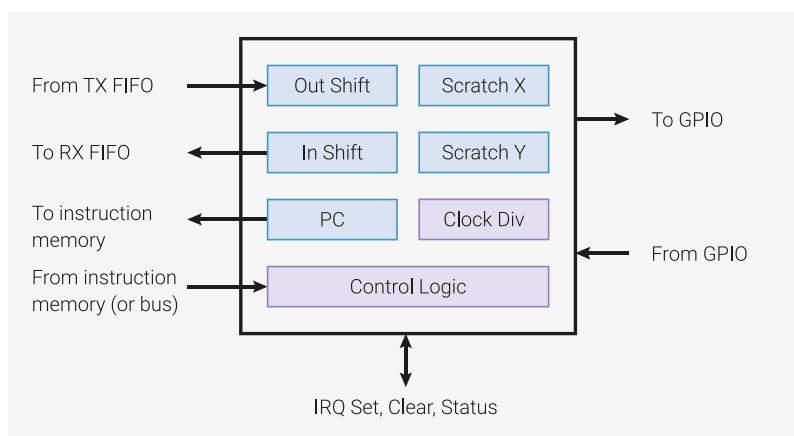
3.2. 程序员模型

四个状态机共享执行一段指令存储器。系统软件将程序加载到该存储器，配置状态机及IO映射，然后启动状态机运行。PIO程序来源多样：用户直接汇编、来自PIO库，或由用户软件编程生成。

从此点开始，状态机通常是自主运行的，系统软件通过DMA、中断和控制寄存器与之交互，如同RP2040上的其他外设。对于更复杂的接口，PIO提供了一套小巧而灵活的原语，允许系统软件更直接地控制状态机的执行流程。

图 39：状态机概述。
数据通过一对FIFO流入和流出。
状态机执行程序，在这些FIFO、一组内部寄存器与引脚之间传输数据。

时钟分频器可将状态机的执行速度按固定比例降低。



3.2.1. PIO 程序

PIO状态机执行简短的二进制程序。

常用接口如UART、SPI或I2C的程序已包含于PIO库中，因此在许多情况下无需编写新的PIO程序。然而，PIO在直接编程时具有更高的灵活性，支持更广泛的

接口类型，这些接口可能并未被其设计者预见。

PIO 总共有九条指令：`JMP`、`WAIT`、`IN`、`OUT`、`PUSH`、`PULL`、`MOV`、`IRQ` 和 `SET`。有关这些指令的详细信息，请参见第 3.4 节。

尽管 PIO 仅包含九条指令，但手工编辑 PIO 程序的二进制文件仍十分困难。PIO 汇编是一种文本格式，用以描述 PIO 程序，其中每条指令对应输出二进制文件中的一条机器指令。以下为 PIO 汇编示例程序：

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> 第 8 至 13 行

```
8 .program squarewave
9     set pindirs, 1      ; 设置引脚为输出
10 again:
11     set pins, 1 [1] ; 将引脚置为高电平，随后延迟一个周期
12     set pins, 0      ; 将引脚置为低电平
13     jmp again        ; 将程序计数器设置为标签 `again`
```

PIO 汇编器包含于 SDK 中，名称为 `pioasm`。该程序处理 PIO 汇编输入文本文件，该文件可能包含多个程序，并输出已组装且可用的程序。对于 SDK，这些组装好的程序以 C 头文件形式输出，包含常量数组：详情请参见第 3.3 节。

3.2.2. 控制流

每个系统时钟周期中，每个状态机都会获取、解码并执行一条指令。每条指令严格占用一个周期，除非指令明确暂停（例如 `WAIT` 指令）。指令还可在执行下一条指令前插入最多 31 个周期的延迟，以辅助编写周期精确的程序。

程序计数器（`PC`）指向本周期正在执行的指令内存地址。通常，`PC` 每周期递增 1，指令内存末端则回绕。跳转指令为例外，它们明确指定 `PC` 的下一取值。

我们的示例汇编程序（如上所示，标记为 `.program squarewave`）展示了这两个概念的实际应用。它驱动一个 50/50 占空比的方波输出到 GPIO，周期为四个时钟周期。利用其他特性（例如 side-set），该周期可缩短至两个时钟周期。

注意

Side-set 指状态机在执行指令的主要副作用之外，同时驱动少量 GPIO 的功能。详细描述见第 3.5.1 节。

系统对指令存储器具有写权限，仅用于加载程序：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 34 行至第 38 行

```
34 // 将汇编程序直接加载至 PIO 的指令存储器。
35 // 每个 PIO 实例拥有 32 槽位指令存储器，4 个状态机均可访问。
36 // 系统仅对其具有写权限。
37 for (uint i = 0; i < count_of(squarewave_program_instructions); ++i)
38     pio->instr_mem[i] = squarewave_program_instructions[i];
```

时钟分频器通过一个常数因子减缓状态机的执行速度，该因子以 16.8 定点小数表示。以上述示例为例，若设置时钟分频为 2.5，则方波的周期将以时钟周期数计算。此功能对于设置串行接口（如 UART）的精确波特率尤为重要。

$$4 \times 2.5 = 10$$

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第42至47行

```

42 // 配置状态机0以sysclk/2.5的速度运行。状态机可
43 // 以每个时钟周期执行一条指令的最快速度运行，但我们可以统一
44 // 降低其速度以满足特定的精确频率需求，例如
45 // UART 波特率。该寄存器具有 16 位整数除数位和 8 位
46 // 小数除数位。
47 pio->sm[0].clkdiv = (uint32_t)(2.5f * (1 << 16));

```

上述代码片段属于完整示例代码，该示例通过 GPIO 0（或其他任意选择映射的引脚）输出12.5MHz方波。我们还可以使用pinsWAIT PIN指令暂停状态机执行一段时间，或使用JMP PIN指令依据引脚状态进行跳转，使控制流因引脚状态而异。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 51 至 59 行

```

51 // 引脚映射分为五组 (out、in、set、side-set、jmp pin)
52 // 这些在不同指令或不同情况下使用。
53 // 此处仅使用 SET 指令。配置状态机 0 的 SET 指令
54 // 仅作用于 GPIO 0；随后配置 GPIO0 由 PIO0 控制，
55 // 而非例如处理器。
56 pio->sm[0].pinctrl =
57     (1 << PIO_SM0_PINCTRL_SET_COUNT_LSB) |
58     (0 << PIO_SM0_PINCTRL_SET_BASE_LSB);
59 gpio_set_function(0, pio_get_funcsel(pio));

```

系统可通过 CTRL 寄存器随时启动或停止各状态机。多个状态机可同时启动，且 PIO 的确定性保证它们能保持精确同步。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 63 至 67 行

```

63 // 启动状态机运行。PIO CTRL 寄存器在 PIO
64 // 实例中为全局，因此您可以启动/停止多个状态机
65 // 同时运行。我们利用寄存器的硬件原子设置别名
66 // 使某一位位高，而无需对寄存器进行读-改-写操作。
67 hw_set_bits(&pio->ctrl, 1 << (PIO_CTRL_SM_ENABLE_LSB + 0));

```

大多数指令从指令存储器执行，但也存在其他来源，且可自由组合：

- 写入特殊配置寄存器 (**SMx INSTR**) 的指令会被立即执行，暂时中断其他指令的执行。例如，写入**SMx INSTR**的**JMP**指令将导致状态机从不同位置开始执行。
- 指令也可通过**MOV EXEC**指令从寄存器执行。
- 指令也可通过**OUT EXEC**指令从输出移位器执行。

最后一个尤其多功能：指令可以嵌入通过FIFO传输的数据流中。

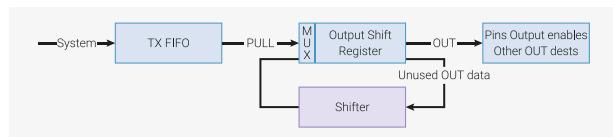
I2C示例利用此功能将例如 **STOP**和 **RESTART**线路状态嵌入到正常数据流中。对于 **MOV**和 **OUT EXEC**，**MOV/OUT**自身在一个周期内执行，执行对象则在下一个周期执行。

3.2.3. 寄存器

每个状态机均包含少量内部寄存器。这些寄存器用于保存输入或输出数据，以及临时值，如循环计数变量。

3.2.3.1. 输出移位寄存器 (OSR)

图40。输出移位寄存器 (OSR)。数据以1至32位为单位发出，未使用的数据通过双向移位器予以回收。OSR一旦为空，即从TX FIFO重新加载数据。



输出移位寄存器 (OSR) 在 TX FIFO 与引脚（或其他目的地，如临时寄存器）之间保存并移位输出数据。

- **PULL** 指令：从 TX FIFO 中移除一个32位字并放入OSR。
- **OUT** 指令将数据从OSR移位至其他目的地，一次移位1至32位。
- 数据移出时，OSR会以零填充。
- 当启用autopull且达到某一总移位计数阈值时，状态机会在执行 **OUT**指令时自动从FIFO重新填充OSR。
- 移位方向可为左或右，由处理器通过配置寄存器进行配置。

例如，以下方法以每两个时钟周期输出一个字节的速率，通过FIFO传输数据至引脚：

```

1 .program pull_example1
2 loop:
3   out pins, 8
4 public entry_point:
5   pull
6   out pins, 8 [1]
7   out pins, 8 [1]
8   out pins, 8
9   jmp loop
  
```

Autopull（参见第3.5.4节）允许硬件在大多数情况下自动重新填充OSR；如果状态机尝试从空的OSR执行 **OUT**操作，则会暂停运行。这带来了两个好处：

- 无需指令在恰当时间显式地从FIFO中拉取数据
- 更高吞吐量：若FIFO保持充足，能在每个时钟周期输出最多32位

配置自动拉取后，上述程序可简化为如下内容，行为完全相同：

```

1 .program pull_example2
2
3 loop:
4   out pins, 8
5 public entry_point:
6   jmp loop
  
```

程序封装（第3.5.2节）允许进一步简化，且可实现每个系统时钟周期输出1字节。

```

1 .program pull_example3
2
3 public entry_point:
4 .wrap_target
5   out pins, 8 [1]
6 .wrap
  
```

3.2.3.2. 输入移位寄存器 (ISR)

图41。输入移位寄存器 (ISR)。
数据一次输入1至32位，当前内容向左或向右移位以留出空间。
填满后，内容会写入接收FIFO。



- **IN** 指令一次将1至32位数据移入寄存器。
- **PUSH** 指令将ISR内容写入接收FIFO。
- ISR 在被推入时会被清零。
- 当启用自动推送功能且达到一定的移位阈值时，状态机会在执行 **IN** 指令时自动将ISR推送。
- 移位方向由处理器通过配置寄存器配置。

某些外设（如UART）必须从左侧移位以获得正确的位序列，因为线路顺序为最低有效位优先；然而，处理器可能期望得到的字节是右对齐的。此问题通过特殊的 **null** 输入源解决，该输入源允许程序员在数据之后向ISR中移入一定数量的零。

3.2.3.3. 移位计数器

状态机会记录通过 **OUT** 指令从OSR移出的位总数，以及通过 **IN** 指令移入ISR的位总数。此信息由一对硬件计数器持续跟踪——输出移位计数器和输入移位计数器——每个计数器均可保存从0至32（含）的数值。每次移位操作时，相应计数器按移位数量递增，最高可达32（等同于移位寄存器的宽度）。状态机可配置为在计数器达到设定阈值时执行特定操作：

- 当移出一定数量的位后，OSR可自动重新填充。详见第3.5.4节。
- 当移入一定数量的位后，ISR可自动清空。详见第3.5.4节。
- **PUSH** 或 **PULL** 指令分别可基于输入或输出移位计数器的状态进行条件执行。

在PIO复位或断言 **CTRL_SM_RESTART** 信号时，输入移位计数器清零（尚未移入任何位），输出移位计数器初始化为32（无剩余位需移出，已完全耗尽）。其他若干指令会影响移位计数器：

- 成功的 **PULL** 操作将输出移位计数器清零。
- 成功的 **PUSH** 操作将输入移位计数器清零。
- **MOV OSR, ...** （即任何写入 **OSR** 的 **MOV** 指令）将输出移位计数器清零为0
- **MOV ISR, ...** （即任何写入 **ISR** 的 **MOV** 指令）将输入移位计数器清零为0
- **OUT ISR, count** 将输入移位计数器设置为 **count**

3.2.3.4. 暂存寄存器

每个状态机均含两个32位内部暂存寄存器，称为 **X** 和 **Y**。

其用途包括：

- **IN/OUT/SET/MOV** 的源或目的地
- 分支条件的源

例如，假设我们希望为“1”数据位产生长脉冲，为“0”数据位产生短脉冲：

```

1 .program ws2812_led
2
3 public entry_point:
4     pull
5     set x, 23      ; 对24位进行循环
6 bitloop:
7     set pins, 1      ; 将引脚设为高电平
8     out y, 1 [5]    ; 将位向右移1位，并写入到y
9     jmp !y skip     ; 如果该位为0，则跳过额外延迟
10    nop [5]
11 skip:
12    设置引脚, 0 [5]
13    jmp x-- bitloop ; 如果x非零则跳转，并将x递减
14    jmp entry_point

```

此处 **X**用作循环计数器，**Y**用作分支时用于读取OSR中单个位的临时变量。该程序可用于驱动WS2812 LED接口，尽管存在更紧凑的实现方案（最少3条指令）。

MOV指令允许使用临时寄存器保存或恢复移位寄存器，以便例如重复移出相同序列。

i 注意

更紧凑的WS2812示例（共4条指令）见第3.6.2节

3.2.3.5. FIFO队列

每个状态机配备一对深度为4字的FIFO队列，分别用于系统向状态机的数据传输（TX）和状态机向系统的数据传输（RX）。TX FIFO由系统总线主控设备（如处理器或DMA控制器）写入，RX FIFO由状态机写入。FIFO实现了PIO状态机与系统总线的时序解耦，使状态机能够在更长时间内无需处理器干预而持续运行。

FIFO还会产生数据请求（DREQ）信号，允许系统DMA控制器根据RX FIFO中数据的存在或TX FIFO中可用空间的情况调整其读写节奏。这样，处理器可以设置一个可能涉及数千字节数据的长事务，事务执行过程中无需进一步的处理器干预。

通常状态机仅在单向进行数据传输。在此情况下，**SHIFTCTRL_FJOIN**选项可以将两个FIFO合并为单向的8级FIFO。此功能对于DPI等高带宽接口尤为重要。

3.2.4. 阻塞

状态机可能因多种原因而暂时暂停执行：

- 等待 **WAIT**指令的条件尚未满足
- 当TX FIFO为空时的阻塞 **PULL**，或当RX FIFO已满时的阻塞 **PUSH**
- 设置了IRQ标志且正在等待其清除的**IRQ WAIT**指令
- 启用自动拉取（**autopull**）且OSR已达到移位阈值时的 **OUT** 指令
- 启用自动推送（**autopush**），ISR达到移位阈值且RX FIFO已满时的 **IN** 指令

在此情况下，程序计数器不会前进，状态机将在下一周期继续执行该指令。若指令指定下一条指令开始前的延迟周期数，则该延迟将于阻塞解除后开始。

ⓘ 注意

侧置设置（第3.5.1节）不受阻塞影响，始终在附属指令的第一个周期执行。

3.2.5. 引脚映射

PIO可控制最多32个GPIO的输出电平和方向，并可检测其输入电平。在每个系统时钟周期内，每个状态机可执行以下操作中的零项、一项或两项：

- 通过 `OUT` 或 `SET` 指令更改部分GPIO的电平或方向，或通过 `IN` 指令读取部分GPIO
- 通过副设置操作更改部分GPIO的电平或方向

上述每项操作均作用于四个连续GPIO范围之一，该范围的起始地址和数量由各状态机的 `PINCTRL` 寄存器配置。每种操作——`OUT`、`SET`、`IN` 及副设置——各对应一个范围。

每个范围可覆盖特定PIO模块可访问的任意GPIO（RP2040上为30个用户GPIO），且范围可相互重叠。

对于每个独立GPIO的输出（电平和方向分别处理），PIO会综合当周期内的最多8次写操作，应用编号最高的状态机的写入。若同一状态机在同一GPIO同时执行 `SET /OUT` 和副设置，则采用副设置操作。如果没有状态机写入此GPIO输出，其值将保持上一个周期的状态不变。

通常，每个状态机的输出映射到一组独立的GPIO，以实现某种外设接口。

3.2.6. IRQ 标志

IRQ标志是状态位，可以由状态机或系统设置或清除。共有8个：全部8个对所有状态机可见，且低4位可通过 `IRQ0_INTE` 和 `RQ1_INTE` 控制寄存器掩码至PIO的中断请求线上。

它们具有两个主要用途：

- 由状态机程序断言系统级中断，并可选择性地等待中断被确认
- 同步两个状态机之间的执行

状态机通过 `IRQ` 和 `WAIT` 指令与这些标志进行交互。

3.2.7. 状态机之间的交互

指令存储器实现为1写4读寄存器文件，因此所有四个状态机可在同一周期读取指令，无需停顿。

多状态机的应用方式有三种：

- 将多个状态机指向同一程序
- 将多个状态机指向不同程序
- 使用多个状态机运行同一接口的不同部分，例如UART的发送（TX）与接收（RX）端，或DPI显示上的时钟/行同步信号与像素数据

状态机之间无法传递数据，但可通过IRQ标志进行同步。共有8个标志（低四位可屏蔽，用作系统IRQ），每个状态机可通过 `IRQ` 指令设置或清除任一标志，并可利用 `WAIT IRQ` 指令等待标志变高或变低。此机制实现了状态机之间的周期精确同步。

3.3. PIO 汇编器 (pioasm)

PIO汇编器解析PIO源文件，输出已汇编的版本，供包含于RP2040应用程序中使用。包括基于SDK构建的C和C++应用程序，以及运行于RP2040 MicroPython移植版的Python程序。

本节简要介绍可用于 `pioasm` 输入的指令及指示。关于如何使用 `pioasm`、其在SDK构建系统中的集成方式、扩展功能（如代码穿透）以及其支持的多种输出格式的深入讨论，请参见[Raspberry Pi Pico系列C/C++ SDK手册](#)。

3.3.1. 指令

以下指令用于控制PIO程序的汇编：

表363. `pioasm`
指令

<code>.define (PUBLIC)<symbol><value></code>	定义一个名为 <code><symbol></code> 的整型符号，其值为 <code><value></code> （ 参见第3.3.2节 ）。若此 <code>.define</code> 位于输入文件中第一个程序之前，则定义对所有程序全局有效，否则仅对定义所在的程序局部有效。若指定了 <code>PUBLIC</code> ，该符号将被包含在汇编输出中，以供用户代码使用。对于SDK，其形式如下：
<code>#define <program_name>_<symbol> value</code>	用于程序符号，或 <code>#define <symbol> value</code> 用于全局符号
<code>.program <name></code>	启动一个名为 <code><name></code> 的新程序。请注意，该名称用于代码中，应为字母数字或下划线组成且不能以数字开头。程序将持续，直到遇到另一个 <code>.program</code> 指令或源文件结束。PIO 指令仅允许在程序内使用。
<code>.origin <offset></code>	可选指令，用于指定程序必须加载到的PIO指令存储偏移地址。此指令最常用于必须从偏移量0加载的程序，因为它们使用基于数据的JMP，其跳转目标（绝对地址）仅存储在少数几位中。此指令在程序外部无效。
<code>.side_set <count> (opt) (pindirs)</code>	如果存在此指令， <code><count></code> 表示所使用的边置（side-set）位数。此外， <code>opt</code> 可用于指示指令中的边置 <code><value></code> 为可选（注意，这需要额外占用一位 —— 除了 <code><count></code> 位之外 —— 从指令延迟的可用位中借用）。最后， <code>pindirs</code> 可用于指定侧集值应应用于PINDIRs而非PINs。此指令仅在程序内且首条指令之前有效。
<code>.wrap_target</code>	置于指令之前，该指令指定程序因环绕而继续执行的指令位置。此指令在程序外无效，程序内仅可使用一次；若未指定，默认指向程序起始位置。
<code>.wrap</code>	置于指令之后，该指令指定在正常控制流（即条件为假时的 <code>jmp</code> 或无 <code>jmp</code> ）下，程序环绕至 <code>.wrap_target</code> 指令之前的位置。此指令在程序外无效，程序内仅可使用一次；若未指定，默认指向最后一条程序指令之后。
<code>.lang_opt <lang> <name> <option></code>	指定与特定语言生成器相关的程序选项。 (参见语言生成器) 此指令在程序外部无效。
<code>.word <value></code>	将一个原始16位数值作为程序中的指令存储。该指令在程序外无效。

3.3.2. 值

以下类型的数值可用于定义整数或分支目标：

表364。pioasm 中的数值，即 `<value>`

整数	整数值，例如 3 或 -7
十六进制	十六进制数值，例如 <code>0xf</code>
二进制	二进制数值，例如 <code>0b1001</code>
符号	由 <code>.define</code> 定义的值（参见 <code>pioasm_define</code> ）
<code><label></code>	标签在程序中的指令偏移量。通常与 JMP 指令一同使用时最为合理（参见第 3.4.2 节）
<code>(<expression>)</code>	待求值的表达式；详见表达式。请注意，括号为必需。

3.3.3. 表达式

表达式可自由用于 pioasm 的值中。

表 365。
pioasm 中的表达式，即 `<expression>`

<code><expression> + <expression></code>	两个表达式之和
<code><expression> - <expression></code>	两个表达式之差
<code><expression> * <expression></code>	两个表达式的乘积
<code><expression> / <expression></code>	两个表达式的整数除法
<code>- <expression></code>	另一个表达式的取反
<code>:: <expression></code>	另一个表达式的位反转
<code><value></code>	任意数值（参见第3.3.2节）

3.3.4. 注释

支持使用 `//` 或 `;进行行注释`

支持使用 `/*` 和 `*/` 进行 C 风格块注释

3.3.5. 标签

标签格式如下：

`<symbol>:`

或

`PUBLIC <symbol>:`

位于行首。

💡 提示

标签实际上是一个自动生成的 `.define`，其值设为当前程序指令偏移量。PUBLIC标签以与PUBLIC.define相同的方式向用户代码公开。

3.3.6. 指令

所有pioasm指令遵循以下通用格式：

`<instruction> (side <side_set_value>) ([<delay_value>])`

其中：

`<instruction>` 为后续章节详细说明的汇编指令。（参见第3.4节）

`<side_set_value>` 为指令开始时应用于side_set引脚的数值（参见第3.3.2节）。请注意，通过`side <side_set_value>`设置的side-set值规则，取决于程序的`.side_set`（参见pioasm_side_set）指令。如果未指定`side_set`，则侧面`<side_set_value>`无效；如果指定为可选数量的 sideset 引脚，则侧面`<side_set_value>`可能存在；如果指定为非可选数量的 sideset 引脚，则侧面`<side_set_value>`是必需的。`<side_set_value>`必须在`.side_set`中指定的 sideset 位数范围内。

指令。

`<delay_value>` 指定指令执行完成后的延迟周期数。delay_value 以数值形式指定（详见第 3.3.2 节），通常取值范围为 0 至 31（含）（5 位值），但启用`.side_set`（详见 pioasm_side_set）指令时，位数会相应减少。如果未提供`<delay_value>`，则指令无延迟。

💡 注意

pioasm 的指令名称、关键字及指令对大小写不敏感；下文的汇编语法部分使用小写字母，因为这符合SDK的风格。

💡 注意

部分汇编语法章节中出现逗号，但完全可选，例如：out pins, 3 可写作out pins 3，jmp x-- /label/ 可写作jmp x--, /label/。下文汇编语法部分均采用第一种风格，因为这符合SDK的用法。

3.3.7. 伪指令

pioasm目前提供一条便捷的伪指令：

`nop`汇编为`mov y, y`。“无操作”指令无特殊副作用，但常用作side-set操作或额外延迟的工具。

3.4. 指令集

3.4.1. 概要

PIO指令长度为16位，其编码如下：

表366. PIO
指令编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
JMP	0	0	0	延迟/副控					条件			地址					
等待	0	0	1	延迟/副控					极性	来源		索引					
输入	0	1	0	延迟/副控					来源			位数					
输出	0	1	1	延迟/副控					目标			位数					
推送	1	0	0	延迟/副控					0	IfF	块	0	0	0	0	0	
拉取	1	0	0	延迟/副控					1	IfE	块	0	0	0	0	0	
MOV	1	0	1	延迟/副控					目标			操作		来源			
IRQ	1	1	0	延迟/副控					0	清除	等待	索引					
设置	1	1	1	延迟/副控					目标			数据					

所有 PIO 指令均在一个时钟周期内完成执行。

5 位延迟/侧置字段的功能取决于状态机的SIDESET_COUNT 配置：

- 最多 5 个最低有效位（5 减去SIDESET_COUNT）用于编码在本指令与下一指令之间插入的空闲周期数。
- 最多 5 个最高有效位，由SIDESET_COUNT设置，用于编码侧置（第 3.5.1 节），可在主指令执行的同时向部分 G PIO 置入常量。

3.4.2. JMP

3.4.2.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	延迟/副控					条件			地址				

3.4.2.2. 操作

如果条件为真，则将程序计数器设置为地址，否则无操作。

延迟周期在 JMP 指令中始终生效，无论条件为真还是假，且发生在条件被评估及程序计数器更新之后。

- 条件：
 - 000: (无条件)：始终成立
 - 001: !X: 寄存器X 为零
 - 010: X--: 寄存器X 非零，递减前
 - 011: !Y: 寄存器Y 为零
 - 100: Y--: 寄存器Y 非零，递减前
 - 101: X!=Y: 寄存器X 不等于寄存器Y
 - 110: PIN: 根据输入引脚分支
 - 111: !OSRE: 输出移位寄存器非空
- 地址：跳转的指令地址。在指令编码中，该地址为PIO指令存储器内的绝对地址。

JMP PIN根据`EXECCTRL JMP_PIN`选择的GPIO进行分支，该配置字段从最多32个状态机可见的GPIO输入中选择一个，独立于状态机的其他输入映射。当GPIO为高电平时，分支被采纳。

!OSRE 比较自上次 **PULL**以来移出的位与由`SHIFTCTRL_PULL_THRESH`配置的移位计数阈值。
该阈值与自动拉取（第3.5.4节）所用阈值相同。

JMP X--和**JMP Y--**分别始终递减暂存寄存器X或Y的值。递减操作不依赖于暂存寄存器当前的值。分支条件基于寄存器的初始值，即递减前的值：若寄存器初始非零，则分支成立。

3.4.2.3. 汇编器语法

`jmp (<cond>) <target>`

其中：

<cond> 为上述可选条件之一（例如 `!x` 表示暂存寄存器X为零）。若未指定条件码，则分支总是被采纳。

<target> 为程序标签或值（参见第3.3.2节），表示程序内指令偏移（第一条指令偏移为0）。请注意，因PIO JMP指令使用PIO指令存储器中的绝对地址，JMP指令需根据程序运行时的加载偏移进行调整。SDK加载程序时会自动处理此问题，但编码供`OUT EXEC`使用的JMP指令时应格外注意。

3.4.3. WAIT

3.4.3.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
等待	0	0	1	延迟/副控				极性	来源		索引					

3.4.3.2. 操作

暂停执行，直到满足特定条件。

如所有暂停指令（第3.2.4节所述）一样，延迟周期自指令完成后开始计数。换言之，若存在延迟周期，其计数不会在等待条件满足之前开始。

- 极性：
 - 1: 等待值为1。
 - 0: 等待值为0。
- 来源：等待的对象。取值如下：
 - 00: **GPIO**: 由 `Index`指定的系统GPIO输入。该GPIO索引为绝对值，不受状态机输入IO映射影响。
 - 01: **PIN**: 由 `Index`选择的输入引脚。该状态机的输入IO映射先行应用，随后由 `Index`选择映射位中等待的具体位。换言之，选定的引脚是通过将`Index`加至`PINCTRL_IN_BASE`配置，取模32计算得出。
 - 10: **IRQ**: 由 `Index`选择的PIO IRQ标志

- 11: 保留
 - Index: 要检查的引脚或位。
- WAIT x IRQ** 的行为与其他 **WAIT** 源存在细微差异：
- 若 **Polarity** 为 1，则状态机在等待条件满足时清除所选的 IRQ 标志。
 - 标志索引的解码方式与 **IRQ** 索引字段一致：若最高有效位（MSB）被设置，则状态机 ID（0...3）通过对最低两位执行模4加法的方式加到 IRQ 索引上。例如，状态机 2 对于标志值“0x11”将等待标志 3，标志值“0x13”将等待标志 1。此功能允许多个运行相同程序的状态机实现相互同步。

⚠ 注意

WAIT 1 IRQ x 不应与提交给中断控制器的 IRQ 标志一起使用，以避免与系统中断处理程序产生竞态条件。

3.4.3.3. 汇编语法

```
wait <polarity> gpio <gpio_num>
wait <polarity> pin <pin_num>
wait <polarity> irq <irq_num> ( rel )
```

其中：

<polarity>	为一个值（详见第3.3.2节），指定极性（0或1）
<pin_num>	为一个值（详见第3.3.2节），指定输入引脚号（根据SM输入引脚映射）
<gpio_num>	为一个值（详见第3.3.2节），指定实际GPIO引脚号
<irq_num> (rel)	为一个值（详见第3.3.2节），指定等待的IRQ编号（0-7）。如果 rel 存在，则实际使用的 irq 号通过将 irq 号 (irq_num_{10}) 的低两位替换为和 ($irq_num_{10} + sm_num_{10}$) 低两位计算得出，其中 sm_num_{10} 为状态机编号。 数字

3.4.4. IN

3.4.4.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
输入	0	1	0		延迟/副控				来源						位数	

3.4.4.2. 操作

将位数位从源移入输入移位寄存器（ISR）。每个状态机的移位方向由 **SIFTCTRL_IN_SHIFTDIR** 配置。此外，将输入移位计数增加位数，最大值饱和为 32。

- 来源：
 - 000: 引脚
 - 001: X (临时寄存器 X)

- 010: **Y** (临时寄存器 Y)
 - 011: 无效 (全零)
 - 100: 保留
 - 101: 保留
 - 110: **ISR**
 - 111: **OSR**
- 位数: 要移入 ISR 的比特数。取值范围为 1 至 32, 32 以 **00000** 编码。

如果启用了自动推送，**IN**将在达到推送阈值 (**SHIFTCTRL_PUSH_THRESH**) 时，将ISR内容推送至RX FIFO。无论是否发生自动推送，IN均在一个周期内执行。自动推送发生时若RX FIFO已满，状态机将进入阻塞状态。自动推送会将ISR内容清零，并重置输入移位计数。详见第3.5.4节。

IN始终使用源数据的最低有效位 (**Bit count** 位)。例如，若PINCTRL_IN_BASE设置为5，指令**IN PINS, 3**将采集第5、6、7号引脚的值，并将其移入ISR。首先ISR向左或向右移位以腾出存放新输入数据的空间，随后将输入数据复制至该空隙。输入数据的位序不依赖于移位方向。

NULL 可用于对ISR内容进行移位。例如，UART接收最低有效位 (LSB) 优先，因此必须向右移位。

经过8次输入引脚，1条指令后，输入的串行数据将占据ISR的31...24位。一次输入**NULL**，24条指令将移入24个零位，使输入数据对齐至ISR的7...0位。或者，处理器或DMA可以从FIFO地址 + 3执行字节读取，该操作将获取FIFO内容的31...24位。

3.4.4.3. 汇编语法

in <source>, <bit_count>

其中：

<*source*> 为上述指定的某一源。

<*bit_count*> 为一个值（参见第3.3.2节），指定移位的位数（有效范围为1至32位）

3.4.5. OUT

3.4.5.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
输出	0	1	1	延迟/副控				目标			位数					

3.4.5.2. 操作

Shift **Bit count** 位从输出移位寄存器 (OSR) 移出，并将这些位写入 **Destination**。此外，输出移位计数增加 **Bit count**，最大饱和至32。

- 目的地：

- 000: 引脚
- 001: **X** (临时寄存器 X)
- 010: **Y** (临时寄存器 Y)

- 011: **NULL** (丢弃数据)
- 100: **PINDIRS**
- 101: **PC**
- 110: **ISR** (同时将ISR移位计数器设为**Bit count**)
- 111: **EXEC** (将OSR移位数据作为指令执行)
- **Bit count**: 要从OSR移出的位数，范围为1至32位，32位编码为**00000**。

一个32位值被写入**Destination**: 低位的**Bit count**位取自OSR，其余位填充为零。该值为OSR的最低有效 **Bit count**位（当**SHIFTCTRL_OUT_SHIFTDIR**为向右时），否则为最高有效位。

PINS 和 **PINDIRS**采用第3.5.6节所述的 **OUT**引脚映射。

如果启用自动拉取，当达到拉取阈值**SHIFTCTRL_PULL_THRESH**时，OSR会自动从TX FIFO重新填充。输出移位计数同时被清零为0。在此情况下，如TX FIFO为空，**OUT**操作将暂停，否则仍在周期内执行。具体内容详见第3.5.4节。

OUT EXEC允许指令以内联形式包含在FIFO数据流中。该 **OUT**本身在一个周期内执行，而OSR中的指令则在下一个周期执行。此机制执行的指令类型无限制。初始 **OUT**指令的延迟周期将被忽略，但被执行指令可正常插入延迟周期。

OUT PC 行为相当于无条件跳转至从OSR移出的地址。

3.4.5.3 汇编语法

out <destination>, <bit_count>

其中：

<destination> 为上述指定的目的地之一。

<bit_count> 为一个值（参见第3.3.2节），指定移位的位数（有效范围为1至32位）

3.4.6. PUSH

3.4.6.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
推送	1	0	0	延迟/副控				0	IfF	块	0	0	0	0	0	0

3.4.6.2. 操作

将ISR内容作为单个32位字推入RX FIFO。将ISR清零。

- **IfFull**: 如果为1，除非总输入移位计数达到其阈值**SHIFTCTRL_PUSH_THRESH**（与自动推送相同；详见第3.5.4节），否则不执行任何操作。
- **Block**: 如果为1，当RX FIFO已满时暂停执行。

PUSH IFFULL有助于使程序更紧凑，类似于自动推送。当启用自动推送导致 **IN**指令在不适当时间阻塞时，此功能尤其有用，例如此时状态机正在断言某些外部控制信号。

PIO汇编器默认设置 `Block`位。若未设置 `Block`位，`PUSH`指令在RX FIFO已满时不会阻塞，而是立即继续执行下一条指令。发生此情况时，FIFO的状态和内容保持不变。ISR仍被清零，且`FDEBUG_RXSTALL`标志被设置（与阻塞状态的`PUSH`或自动推送（到已满的RX FIFO）相同），以指示数据丢失。

3.4.6.3. 汇编语法

```
push ( iffull )
push ( iffull ) block
push ( iffull ) noblock
```

说明：

- `iffull` 等同于上述的`IfFull == 1`。即未指定时默认为`IfFull == 0`。
- `block` 等同于上述的`Block == 1`。若未指定`block`或`noblock`，则此项为默认。
- `noblock` 等同于上述的`Block == 0`。

3.4.7. PULL

3.4.7.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
拉取	1	0	0	延迟/副控				1	IfE	块	0	0	0	0	0	0

3.4.7.2. 操作

从TX FIFO加载一个32位字至OSR。

- `IfEmpty`：若为1，则除非总输出移位计数达到阈值`SHIFTCTRL_PULL_THRESH`（与`autopull`相同；详见第3.5.4节），否则不执行任何操作。
- `Block`：若为1，则当TX FIFO为空时暂停执行。若为0，从空FIFO中读取时，将寄存器X的内容复制到OSR。

某些外设（UART、SPI等）在无数据时应暂停，并在数据到达时继续处理；其他外设（I2S）应持续时钟运行，输出占位符或重复数据优于停止时钟。此功能可通过`Block`参数实现。

在空FIFO上执行非阻塞`PULL`，其效果等同于`MOV OSR, X`指令。程序可预先将寄存器X加载合适默认值，或在每次`PULL NOBLOCK`之后执行`MOV X, OSR`，以便在新数据到达前重复使用最后一条有效FIFO数据。

当TX FIFO为空且自动拉取（`autopull`）的`OUT`操作将在不适当的位置停顿时，`PULL IFEMPTY`指令非常有用。`IfEmpty`允许实现与自动拉取相同的程序简化，例如消除外层循环计数器，但停顿会发生在程序的受控位置。

ⓘ 注意

启用自动拉取 (autopull) 时，当OSR已满，任何 **PULL** 指令均为无操作，因此 **PULL** 指令表现为屏障。**OUT NULL, 32** 可用于显式丢弃OSR内容。详见第3.5.4.2节。

3.4.7.3 汇编器语法

pull (ifempty)

pull (ifempty) 块

pull (ifempty) 非块

说明：

ifempty 相当于前述的 **IfEmpty == 1**。即若未指定，默认条件为 **IfEmpty == 0**。

block 等同于上述的 **Block == 1**。若未指定 **block** 或 **noblock**，则此项为默认。

noblock 等同于上述的 **Block == 0**。

3.4.8. MOV

3.4.8.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	延迟/副控				目标			操作		来源			

3.4.8.2. 操作

从 **Source** 复制数据至 **Destination**。

• 目的地：

- 000: **PINS** (使用与 **OUT** 相同的引脚映射)
- 001: **X** (临时寄存器 X)
- 010: **Y** (临时寄存器 Y)
- 011: 保留
- 100: **EXEC** (将数据作为指令执行)
- 101: **PC**
- 110: **ISR** (该操作将输入移位计数器重置为 0，即清空)
- 111: **OSR** (该操作将输出移位计数器重置为 0，即填满)

• 操作：

- 00: 无
- 01: 反转 (按位取反)
- 10: 位反转
- 11: 保留

- 来源：
 - 000: PINS (使用与 IN相同的引脚映射)
 - 001: X
 - 010: Y
 - 011: NULL
 - 100: 保留
 - 101: 状态
 - 110: ISR
 - 111: OSR

MOV PC会导致无条件跳转。**MOV EXEC**的行为与**OUT EXEC**（第3.4.5节）相同，允许将寄存器内容作为指令执行。**MOV**本身执行1个周期，下一周期执行Source中的指令。**MOV EXEC**上的延迟周期会被忽略，但被执行指令可正常插入延迟周期。

STATUS源的值为全1或全0，取决于某些状态机状态（如FIFO已满/已空），由EXECCTRL_STATUS_SEL配置。

MOV可对传输的数据进行有限的操作，具体由 **Operation**参数指定。反转操作将目标寄存器Destination中的每个位设为对应源寄存器Source位的逻辑非，即1变0，0变1。位逆操作将目标寄存器Destination中的每个位n设为源寄存器Source中第31-n位的值，假定位编号为0至31。

MOV dst, PINS 使用 IN引脚映射读取引脚，并将完整的32位值写入目标寄存器，无需掩码。

读取值的最低有效位（LSB）对应PINCTRL_IN_BASE指定的引脚，后续的每个位来自编号更高的引脚，超过31时回绕。

3.4.8.3. 汇编语法

`mov <destination>, (op) <source>`

说明：

<destination> 为上述指定的目的地之一。

<op> 若存在，格式为：

! 或 ~表示逻辑非（注意：始终为按位非）

:: 表示位反转

<source> 为上述指定的某一源。

3.4.9. IRQ

3.4.9.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	延迟/副控				0	清除	等待	索引					

3.4.9.2. 操作

设置或清除由 `Index` 参数指定的IRQ标志。

- 清除：若为1，则清除由 `Index` 指定的标志，而非置位。如果 `Clear` 被设置，则 `Wait` 位无效。
- 等待：若为1，则暂停执行，直到被置位的标志被清除，例如系统中断处理程序已确认该标志。
- 索引：
 - 最低三位（3 LSBs）指定一个IRQ索引，范围为0至7。该IRQ标志将根据清除位设置或清除。
 - 如果最高位（MSB）被置位，则通过对最低两位（2 LSBs）进行模4加法，将状态机ID（0至3）加至IRQ索引。例如，状态机2且标志值为0x11时将触发标志3，标志值为0x13时将触发标志1。

IRQ标志4至7仅对状态机可见；IRQ标志0至3可路由至系统级中断，经过PIO的两个外部中断请求线路中的任一条，该线路由 `IRQ0_INTE` 和 `IRQ1_INTE` 配置。

模加位允许 ‘IRQ’ 与 ‘WAIT’ 指令的相对寻址，以同步运行相同程序的状态机。第2位（第三个最低有效位）不受该加法影响。

如果 `Wait` 被设置，`Delay` 周期将在等待期结束后开始。

3.4.9.3. 汇编语法

```
irq <irq_num> (rel)
irq set <irq_num> (rel)
irq nowait <irq_num> (rel)
irq wait <irq_num> (rel)
irq clear <irq_num> (rel)
```

说明：

`<irq_num> (rel)` 为一个值（详见第3.3.2节），指定等待的IRQ编号（0-7）。如果 `rel` 存在，则实际使用的 `irq` 号通过将 `irq` 号 (irq_num_{10}) 的低两位替换为和 ($irq_num_{10} + sm_num_{10}$) 低两位计算得出，其中 sm_n 为状态机编号。

数字

<code>irq</code>	表示设置IRQ且不等待
<code>irq set</code>	同样表示设置IRQ且不等待
<code>irq nowait</code>	再次表示设置IRQ且不等待
<code>irq wait</code>	表示设置IRQ并等待其被清除后继续
<code>irq clear</code>	表示清除IRQ

3.4.10. SET

3.4.10.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
设置	1	1	1	延迟/副控				目标			数据					

3.4.10.2. 操作

将立即数 **Data** 写入至 **Destination**。

- 目的地：
 - 000: 引脚
 - 001: X (临时寄存器 X) 低 5 位设为 **Data**, 其他位清零。
 - 010: Y (临时寄存器 Y) 低 5 位设为 **Data**, 其他位清零。
 - 011: 保留
 - 100: **PINDIRS**
 - 101: 保留
 - 110: 保留
 - 111: 保留
- **Data**: 用于驱动引脚或寄存器的 5 位立即数。

可用于断言控制信号, 如时钟或芯片选择, 或初始化循环计数器。由于 **Data** 为 5 位, 临时寄存器可通过 **SET** 设为 0 至 31 的值, 足以支持 32 次迭代的循环。

SET 和 **OUT** 指令映射至引脚的方式是独立配置的。它们可映射至不同位置, 例如一个引脚用作时钟信号, 另一个用于数据。也可以是引脚的重叠范围: UART 发送器可能用 **SET** 断言起始和停止位, 使用 **OUT** 指令将 FIFO 数据移出至相同引脚。

3.4.10.3. 汇编语法

set <destination>, <value>

其中:

- | | |
|----------------------------|---------------------------------|
| <destination> | 为上述指定的目的地之一。 |
| <value> | 要设置的值 (参见第3.3.2节), 有效范围为 0 至 31 |

3.5. 功能细节

3.5.1. 旁路设定

Side-set 功能允许状态机在指令主执行的同时, 同时改变最多 5 个引脚的电平或方向。

一个典型应用为高速 SPI 接口: 时钟信号的跳变 (从 1→0 或 0→1) 必须与数据的跳变同步, 即通过 OSR 向 GPIO 移位新的数据位。在此情况下, 带 side-set 的 **OUT** 指令能够同时完成这两项操作。

该功能提高了接口的时序精度, 减小了程序整体尺寸 (无需额外 **SET** 指令切换时钟引脚), 并提升了 SPI 的最高工作频率。

此外, Side-set 使 GPIO 映射更为灵活, 其映射方式独立于 **SET** 指令。示例 I2C 代码允许在禁用时钟拉伸的情况下, 将 SDA 和 SCL 映射到任意两个引脚。通常, SCL 信号用于切换以同步数据传输, 而 SDA 包含被移出的数据位。然而, 一些特定的 I2C 序列, 如 **Start** 和 **Stop** 线条件, 需要在 SDA 及 SCL 上驱动固定的信号模式。I2C 为实现此目的所采用的映射是:

- Side-set → SCL
- OUT → SDA
- SET → SDA

此设计允许状态机同时满足两种使用场景：SDA上传输数据且SCL作为时钟，或在SDA与SCL上产生固定的切换信号，同时仍支持将SDA和SCL映射至任意两个所选GPIO。

side-set数据编码在每条指令的Delay/side-set字段中。任何指令均可与side-set组合，包括向引脚写入的指令，如 OUT PINS 或 SET PINS。边集的引脚映射独立于 OUT 和 SET 映射，尽管可能存在重叠。若边集与 OUT 或 SET 同时写入同一引脚，则以边集数据为准。

注意

即使指令阻塞，边集仍会立即生效。

```

1 .program spi_tx_fast
2 .side_set 1
3
4 loop:
5     输出引脚，1侧 0
6     jmp loop      边 1

```

spi_tx_fast示例表明两大优势：数据与时钟跳变可更精确对齐，且程序整体加速，当前情况下每两个系统时钟周期输出一位。程序也可变得更小。

使用边集时需配置四项参数：

1. 用于边集而非延迟的Delay/side-set字段最高有效位数。由PINCTRL_SIDESET_COUNT进行配置。若设置为5，则无法使用延迟周期。如果设置为0，则不会发生侧设定。
2. 是否使用这些位中的最高有效位作为启用位。只有当启用位为高时，侧设定才会在指令上执行。如果没有启用位，且SIDESET_COUNT非零，则该状态机上的每条指令都会执行侧设定。该配置由EXECCTRL_SIDE_EN进行设置。
3. 映射最低有效侧设定位的GPIO编号。由PINCTRL_SIDESET_BASE进行配置。
4. 侧设定是写入GPIO电平还是GPIO方向。由EXECCTRL_SIDE_PINDIR进行配置。

在上述示例中，只有一个侧设定数据位，且每条指令均执行侧设定，因此不需要启用位。SIDESET_COUNT设置为1，SIDE_EN设置为假。SIDE_PINDIR亦为假，因为我们希望驱动时钟的高低电平，而非高阻和低阻状态。SIDESET_BASE用于选择时钟所驱动的GPIO端口。

3.5.2. 程序封装

PIO程序通常包含“外层循环”：它们重复执行相同的步骤序列，在FIFO与外部世界之间传输数据流。引言中的方波程序即为此类最简示例：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> 第8至13行

```

8 .program squarewave
9     set pindirs, 1      ; 设置引脚为输出
10 again:
11     set pins, 1 [1] ; 将引脚置为高电平，随后延迟一个周期
12     set pins, 0          ; 将引脚置为低电平

```

```
13     jmp again      ; 将程序计数器设置为标签 `again`
```

程序主体将引脚拉高，然后拉低，产生一个完整的方波周期。整个程序循环执行，实现周期性输出。跳转指令本身占用一个周期，每条 `set` 指令亦是如此。为了保持高电平和低电平的时长一致，`set pins, 1` 指令增加了一个延迟周期，使状态机在执行 `set pins, 0` 指令前空闲一个周期。总体而言，每个循环耗时四个周期。此处存在两个不足之处：

- `JMP` 指令占用指令存储空间，减少了可用于其他程序的空间。
- 执行 `JMP` 指令所需的额外周期最终导致最大输出速率降低一半。

由于程序计数器（`PC`）在超过31后自然回绕至0，我们可以通过将整个指令存储器填充为重复的 `set pins, 1` 与 `set pins, 0` 模式来解决第二个问题，但这存在资源浪费。状态机具备通过其 `EXECCTRL` 控制寄存器配置的硬件特性，能够解决此类常见情况。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_wrap.pio 第12至20行

```
12 .program squarewave_wrap
13 ; 类似于squarewave，但使用状态机的.wrap硬件功能替代
14 ; 显式jmp。这是一个免费的（0周期）无条件跳转指令。
15
16     set pindirs, 1      ; 将引脚设置为输出
17 .wrap_target
18     set pins, 1 [1] ; 将引脚置为高电平，随后延迟一个周期
19     set pins, 0 [1] ; 将引脚拉低，然后延迟一个周期
20 .wrap
```

从程序存储器执行指令后，状态机使用以下逻辑更新 `PC`：

1. 如果当前指令为 `JMP`，且 `Condition` 为真，则将 `PC` 设置为 `Target`
2. 否则，若 `PC` 匹配 `EXECCTRL_WRAP_TOP`，则将 `PC` 设置为 `EXECCTRL_WRAP_BOTTOM`
3. 否则，递增 `PC`；若当前值为31，则将其设置为0。

`.wrap_target` 和 `.wrap` 汇编指令在 `pioasm` 中实质上为标签。它们导出可分别写入 `WRAP_BOTTOM` 和 `WRAP_TOP` 控制字段的常量：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/generated/squarewave_wrap.pio.h

```
1 // -----
2 // 本文件由 pioasm 自动生成，禁止编辑！ //
3 // -----
4
5 #pragma once
6
7 #include "hardware/pio.h"
8
9 // -----
10 // squarewave_wrap //
11 // -----
12
13 #define squarewave_wrap_target 1
14 #define squarewave_wrap_wrap 2
15 #define squarewave_wrap_pio_version 0
16
17 static const uint16_t squarewave_wrap_program_instructions[] = {
18     0xe081, // 0: 设置 pindirs, 1
19                 //     .wrap_target
20     0xe101, // 1: 设置 pins, 1           [1]
21     0xe100, // 2: 设置 pins, 0           [1]
```

```

22         //      .wrap
23 };
24
25 static const struct pio_program squarewave_wrap_program = {
26     .instructions = squarewave_wrap_program_instructions,
27     .length = 3,
28     .origin = -1,
29     .pio_version = squarewave_wrap_pio_version,
30     .used_gpio_ranges = 0x0
31 #endif
32 };
33
34 static inline pio_sm_config squarewave_wrap_program_get_default_config(uint offset) {
35     pio_sm_config c = pio_get_default_sm_config();
36     sm_config_set_wrap(&c, offset + squarewave_wrap_target, offset +
37                         squarewave_wrap_wrap);
38 }

```

这是来自PIO汇编器 `pioasm` 的原始输出，生成了一个默认的 `pio_sm_config` 对象，其中包含程序列表中的 `WRAP` 寄存器值。控制寄存器字段也可以直接初始化。

注意

`WRAP_BOTTOM` 和 `WRAP_TOP` 是 PIO 指令存储器中的绝对地址。如果程序以偏移量加载，则必须相应调整包裹地址。

`squarewave_wrap` 示例插入了延迟周期，因此其行为与原始的 `quarewave` 程序完全一致。得益于程序包裹功能，这些延迟现在可以移除，从而使输出的切换速度加倍，同时保持高低周期的均衡。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_fast.pio 第12 - 18行

```

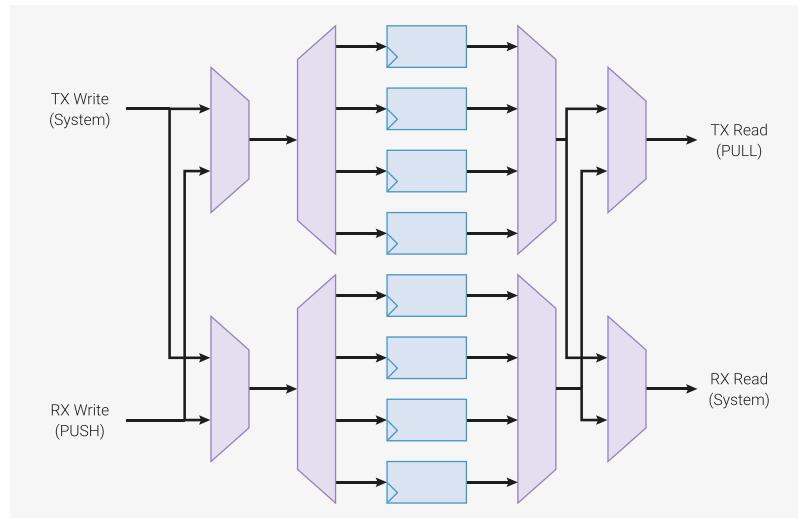
12 .program squarewave_fast
13 ; 类似于 squarewave_wrap，但去除延时周期，因此运行速度加倍。
14     set pindirs, 1      ; 将引脚设置为输出
15 .wrap_target
16     set pins, 1          ; 将引脚置高
17     set pins, 0          ; 将引脚置为低电平
18 .wrap

```

3.5.3. FIFO 连接

默认情况下，每个状态机在每个方向拥有一个4条目FIFO：一条用于系统向状态机传输数据（TX），另一条用于相反方向（RX）。然而，许多应用不需要系统与单个状态机之间的双向数据传输，但可能需要更深的FIFO，特别是在高带宽接口如DPI中。对于这些情况，`SHIFTCTRL_FJOIN` 可将两个4条目FIFO合并为单个8条目FIFO。

图42. 可合并的双FIFO。由四个数据寄存器、一个1:4译码器及一个4:1多路复用器组成的一对四条目FIFO。额外的复用允许写数据和读数据在TX和RX通道之间交叉传输，使所有8个条目均可从两个端口访问。



另一个例子是UART：由于UART的TX/CTS和RX/RTS部分是异步的，故实现于两个独立的状态机。让每个状态机一半的FIFO资源闲置属于极大浪费。将两半合并为仅供TX/CTS状态机使用的TX FIFO，或仅供RX/RTS状态机使用的RX FIFO，实现了资源的充分利用。配备8级FIFO的UART可使中断间隔时间比仅具4级FIFO的延长一倍。

当一个FIFO大小增加（由4增至8）时，同一状态机上的另一个FIFO会被减少至零。例如，如果合并至TX，则RX FIFO不可用，任何 **PUSH** 指令将被阻塞。RX FIFO将在 FSTAT 寄存器中同时显示为 RXFULL 和 RXEMPTY。如果连接至RX，则相反情况成立：TX FIFO不可用，且该状态机的 TXFULL 和 TXEMPTY 位均在 FSTAT 中被置位。同时设置 FJOIN_RX 和 FJOIN_TX 将导致两个FIFO均不可用。

8个FIFO项对于通过RP2040系统DMA每时钟周期传输1个字是足够的，前提是DMA未因与其他主控器竞争而减速。

⚠ 注意

更改 **FJOIN** 会丢弃状态机 FIFO 中存在的所有数据。如该数据不可替代，必须先行清空。

3.5.4. 自动推送与自动拉取

每执行一个 **OUT** 指令，OSR会随着数据移出而逐渐清空。OSR清空后必须重新填充：例如，使用 **PULL** 将一个字的数据从 TX FIFO 传输至 OSR。同理，ISR在填满后也必须清空。一种方法是在移动适量数据后通过循环执行 **PULL** 操作，如下所示：

```

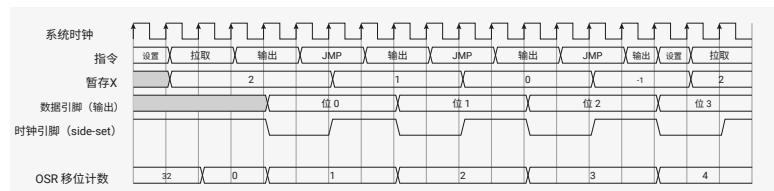
1 .program manual_pull
2 .side_set 1 opt
3
4 .wrap_target
5   set x, 2           ; X = 位数 - 2
6   pull                side 1 [1] ; 如果无TX数据，则在此处停顿
7 bitloop:
8   out pins, 1        side 0 [1] ; 移位输出数据位并将时钟拉低
9   jmp x-- bitloop side 1 [1] ; 循环执行3次
10  out pins, 1       side 0      ; 在重新加载X之前移位输出最后一位
11 .wrap

```

此程序以恒定速率（每4个周期1位）从每个FIFO字输出4位数据，并伴有位时钟。当TX FIFO为空时，程序将在时钟保持高电平状态下停顿（侧边设置side-set仍会在指令停顿周期执行）。

图43展示了状态机如何执行该程序。

图 43。 manual_pul
I 程序的执行。
。 X 用作循环计数器。每次迭代移出一个数据位，时钟先被拉低，随后拉高。每条指令中包含一个延迟周期，使得每次迭代总周期数达到四个。第三次循环后，第四个位被移出，状态机立即返回。程序起始，重新加载循环计数器并拉取新数据，同时保持每位4周期的节奏。



该程序存在以下限制：

- 它占用5个指令槽，但仅2个指令立即有效（out pins, 1条设置为0，另一条设置为1），用于输出串行数据与时钟。
 - 由于拉取新数据和重新加载循环计数器需额外周期，其吞吐量限制为系统时钟频率的四分之一。
- 这是PIO中一种常见的问题类型，因此每个状态机配备了一些额外硬件来进行处理。状态机会跟踪OSR的总移位计数 OUT 和ISR的总移位计数 IN，并在这些计数达到可编程阈值时触发相应操作。
- 当 OUT 指令达到或超过拉取阈值时，若数据可用，状态机可同时从TX FIFO重新填充OSR。
 - 当 IN 指令达到或超过推送阈值时，状态机可将移位结果直接写入RX FIFO，且清除ISR。

manual_pull示例可重写以利用自动拉取（autopull）：

```
1 .program autopull
2 .side_set 1
3
4 .wrap_target
5     out pins, 1    side 0      [1]
6     nop            side 1      [1]
7 .wrap
```

该程序较原始版本更短更简洁，若去除延迟周期，运行速度可提升两倍，因硬件“免费”完成OSR的重新填充。请注意，该程序不会在下一次拉取之前确定总的移位位数；硬件在达到可编程阈值SHIFCTRL_PULL_THRESH后会自动拉取，因此相同程序亦可从每个FIFO字中移出例如16或32位。

最后，请注意，上述程序并非完全等同于原始程序，因为它在时钟输出为低电平时停滞，而非高电平状态。我们可以使用 PULL_IFEMPTY 指令改变停滞位置，该指令采用与自动拉取相同的可配置阈值：

```
1 .program somewhat_manual_pull
2 .side_set 1
3
4 .wrap_target
5     out pins, 1    side 0      [1]
6     pull_ifempty   side 1      [1]
7 .wrap
```

以下是一个完整示例（PIO程序及用于加载和运行该程序的C语言代码），演示在同一个状态机上同时启用自动拉取和自动推送。该程序配置状态机0，将数据从TX FIFO回环至RX FIFO，吞吐率为每两个时钟周期传输一个字。该示例还演示了当OSR和TX FIFO均为空时，状态机尝试执行 OUT 操作会导致停滞的情况。

```

1 .program auto_push_pull
2
3 .wrap_target
4     out x, 32
5     in x, 32
6 .wrap

```

```

1 #include "tb.h" // TODO 本文件基于现有软件架构构建，以支持 printf 等功能
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "auto_push_pull.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     // 加载程序并配置状态机 0，启用自动推送/拉取
15     // 阈值设为 32，并在程序边界处自动回绕。阈值 32
16     // 由寄存器值 00000 编码。
17     for (int i = 0; i < count_of(auto_push_pull_program); ++i)
18         mm_pio->instr_mem[i] = auto_push_pull_program[i];
19     mm_pio->sm[0].shiftctrl =
20         (1u << PIO_SM0_SHIFTCTRL_AUTOPUSH_LSB) |
21         (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB) |
22         (0u << PIO_SM0_SHIFTCTRL_PUSH_THRESH_LSB) |
23         (0u << PIO_SM0_SHIFTCTRL_PULL_THRESH_LSB);
24     mm_pio->sm[0].execctrl =
25         (auto_push_pull_wrap_target << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB) |
26         (auto_push_pull_wrap << PIO_SM0_EXECCTRL_WRAP_TOP_LSB);
27
28     // 启动状态机 0
29     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
30
31     // 向 TX FIFO 推送数据，从 RX FIFO 弹出数据
32     for (int i = 0; i < 5; ++i)
33         mm_pio->txf[0] = i;
34     for (int i = 0; i < 5; ++i)
35         printf("%d\n", mm_pio->rx[0]);
36
37     return 0;
38 }

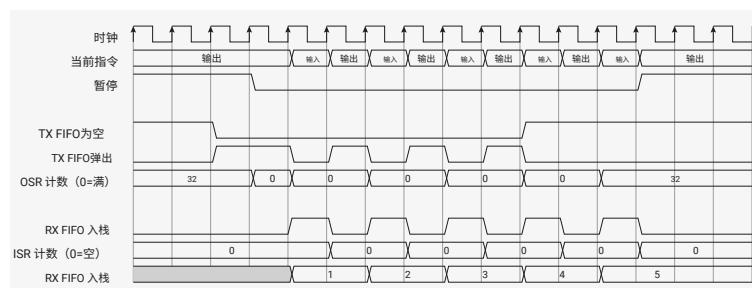
```

图44展示了状态机如何执行示例程序。最初OSR为空，因此状态机在第一个 OUT指令处暂停。一旦TX FIFO中有数据，状态机会将其传输到OSR。在下一周期， OUT可使用OSR中的数据执行（此处为将数据传输至X暂存寄存器），状态机同时从FIFO重新填充OSR。由于每条 IN指令会立即填充ISR，ISR保持为空，且 IN指令直接将数据从X暂存寄存器传输到RX FIFO。

图 44。auto_push_pull 程序的执行。
状态机在 OUT 指令处暂停，直到数据通过 TX FIFO 传输至 OSR。

随后，由于位计数为 32，
OSR 在每次 OUT 操作时同步重新填充，且 IN 数据绕过 ISR 直接

进入 RX FIFO。当 FIFO 被清空时，状态机再次暂停。
OSR 也再次为空。



为了在正确时间触发自动推送或拉取，状态机使用一对饱和 6 位计数器跟踪 ISR 和 OSR 的总移位计数。

- 复位时，或在 `CTRL_SM_RESTART` 断言时，ISR 移位计数器设为 0（无移入数据），OSR 设为 32（无剩余待移出数据）
-
- 一条 `OUT` 指令将 OSR 移位计数器增加位计数。
- 一条 `IN` 指令使 ISR 移位计数器增加比特数
- 一条 `PULL` 指令或自动拉取将 OSR 计数器清零为 0
- 一条 `PUSH` 指令或自动推送将 ISR 计数器清零为 0
- 一条 `MOV OSR, n` 或 `MOV ISR, n` 指令分别将 OSR 或 ISR 移位计数器清零为 0
- 一条 `OUT ISR, n` 指令将 ISR 移位计数器设置为 `n`

在任何 `OUT` 或 `IN` 指令中，状态机将移位计数器与 `SHIFTCTRL_PULL_THRESH` 和 `SHIFTCTRL_PUSH_THRESH` 的值进行比较，以决定是否需要执行相应操作。自动拉取与自动推送分别通过 `SHIFTCTRL_AUTOPULL` 和 `SHIFTCTRL_AUTOPUSH` 字段启用。

3.5.4.1. 自动推送详细说明

启用自动推送功能的“IN”指令伪代码：

```

1 isr = shift_in(isr, input())
2 isr count = saturate(isr count + in count)
3
4 如果 rx 计数 ≥ 阈值：
5   如果 rx FIFO 已满：
6     暂停
7   else:
8     推送(isr)
9   isr = 0
10 _isr 计数 = 0

```

请注意，该硬件在单个机器时钟周期内完成上述步骤（除非发生暂停）。

阈值可配置，范围为 1 至 32。

3.5.4.2. 自动拉取详细信息

在非“OUT”周期内，硬件执行的操作等同于以下伪代码：

```

1 如果 MOV 或 PULL:
2     osr 计数 = 0
3
4 如果 osr 计数 ≥ 阈值:
5     如果 tx FIFO 非空:
6         osr = pull()
7         osr 计数 = 0

```

因此，自动拉取可在两个“OUT”周期之间的任意时刻发生，具体取决于数据到达 FIFO 的时间。

在“OUT”周期内，执行顺序略有不同：

```

1 如果 osr 计数 ≥ 阈值:
2     如果 tx FIFO 非空:
3         osr = pull()
4         osr 计数 = 0
5     暂停
6 else:
7     output(osr)
8     osr = shift(osr, out count)
9     osr count = saturate(osr count + out count)
10
11    if osr count >= threshold:
12        如果 tx FIFO 非空:
13            osr = pull()
14            osr 计数 = 0

```

硬件能够在移出最后一部分移位数据的同时重新填充OSR，因为这两个操作可以并行执行。然而，不能在同一周期内填充空的OSR并执行“OUT”操作，因这将导致较长的逻辑路径。

重新填充与您的程序具有一定的异步性，但“OUT”操作表现为数据屏障，状态机将永远不会“OUT”未写入FIFO的数据。

请注意，当启用autopull时，从OSR执行的“MOV”操作行为未定义；您可能读取到尚未移出残留的数据，或来自FIFO的新字，这取决于与系统DMA的竞争状态。同样，向OSR执行的“MOV”操作可能覆盖刚自动拉取的数据。然而，您通过‘MOV’指令写入OSR的数据将永远不会被覆盖，因为‘MOV’会更新移位计数器。

如果您确实需要读取OSR内容，应执行显式的‘PULL’操作。上述非确定性是硬件自动管理拉取操作所带来的代价。启用自动拉取（autopull）后，‘PULL’指令的行为将改变：如果OSR已满，则该指令不执行任何操作。此设计旨在避免与系统DMA发生竞态条件。其行为类似屏障操作：要么自动拉取已完成，此时‘PULL’无效；要么程序将在‘PULL’指令处阻塞，直至FIFO中出现可用数据。

‘PUSH’无需类似行为，因为自动推送（autopush）不存在相同的非确定性。

3.5.5. 时钟分频器

PIO以系统时钟为基础运行，但对于许多接口而言速度过快，可插入的Delay周期数量有限。某些设备，例如UART，要求精确控制并变化信号速率，理想情况下，多个状态机可在运行相同程序的同时独立变化。为此，每个状态机均配备有时钟分频器。

时钟分频器不是直接降低系统时钟本身，而是重新定义执行时“一个周期”包含的系统时钟周期数。其通过生成时钟使能信号实现，该信号可在每个系统时钟周期内暂停或恢复执行。时钟分频器以固定间隔生成时钟使能脉冲，

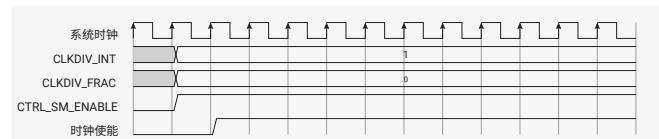
使状态机以某种稳定速率运行，速度可能远低于系统时钟。

通过这种方式实现时钟分频器，使状态机与系统之间的接口更简单、延迟更低且占用资源更小。当时钟使能为低时，状态机完全空闲，但系统仍可访问状态机的 FIFO 并更改其配置。

时钟分频器采用16位整数和8位小数表示，并使用一阶 delta-sigma 进行小数分频。时钟除数可在1至65536之间变化，增量为 $1 / 256$.

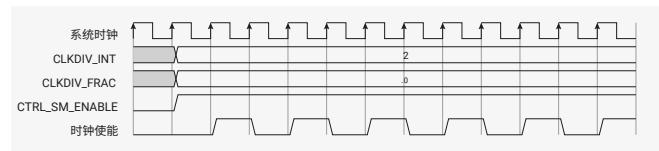
若时钟除数设置为1，则状态机会在每个周期运行，即全速运作：

图45。时钟除数为1时状态机的工作原理。一旦通过CTRL_SMR 寄存器启用状态机，其时钟使能将在每个周期被激活。



通常，整数时钟除数 n 使状态机每 n 周期运行一次，有效时钟频率为 f_{sys} / n .

图46。整数时钟除数产生周期性的时钟使能信号。时钟分频器不断从 n 递减计数，并在

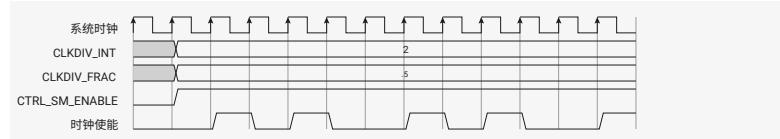


计数至1时发出使能脉冲。分数除法将保持稳定的分频速率为

$n + f / 256$ ，其中 n 与 f 分别表示该状态机 CLKDIV 寄存器中的整数与分数部分。其方式是通过选择性地将某些分频周期从一个周期延长至另一周期

$n + 1$.

图47。平均除数为2.5的分数时钟分频。时钟分频器维护每个分频周期分数值的累积总和，每当该值跨越1时，下一分频周期的整数除数即增加1。



对于小的 n ，分数分频器引入的抖动可能不可接受。然而，对于较大的数值，该效应则显著减弱。

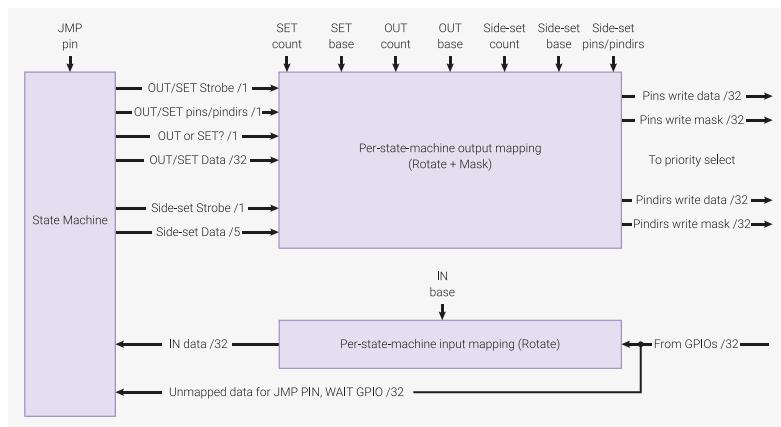
① 注意

对于高速异步串行，建议尽可能使用偶数分频或1 Mbaud的倍数，而非传统的300倍数，以避免不必要的抖动。

3.5.6. GPIO 映射

内部，PIO设有一个32位寄存器用于其可驱动的每个GPIO的输出电平，另有一个寄存器用于输出使能（高电平/高阻态）。在每个系统时钟周期内，每个状态机均可向这些寄存器中的部分或全部GPIO写入数据。

图48。状态机具有两个独立的输出通道，一个由OUT/SET共享，另一个用于side-set（可在任意时间发生）。三个独立映射（首个GPIO及GPIO数量）控制OUT、SET和side-set信号对应的GPIO端口。输入数据根据映射到IN数据最低有效位的GPIO进行循环移位。



用于输出电平和输出使能寄存器的写入数据及写入掩码来源如下：

- 一条 OUT 指令最多写入32位。依据指令的 Destination 字段，写入操作应用于引脚或引脚方向寄存器。OUT 数据的最低有效位映射至 PINCTRL_OUT_BASE，映射范围覆盖 PINCTRL_OUT_COUNT 位，超出 GPIO31 后循环映射。
- 一条 SET 指令最多写入5位。依据指令的 Destination 字段，写入操作应用于引脚或引脚方向寄存器。SET 数据的最低有效位映射至 PINCTRL_SET_BASE，映射范围覆盖 PINCTRL_SET_COUNT 位，超出 GPIO31 后循环映射。
- 侧置操作最多可写入5位。根据寄存器字段 EXECCTRL_SIDE_PDIR，该操作应用于引脚或引脚方向。侧置数据的最低有效位映射至 PINCTRL_SIDESET_BASE，连续映射至 PINCTRL_SIDESET_COUNT 个引脚，若 EXECCTRL_SIDE_EN 被设置，则数量减一。

每个 OUT/SET/侧置操作写入一段连续的引脚范围，但这些范围在32位GPIO空间内的大小与位置均独立设定。这在多种应用场景中具备足够的灵活性。例如，若一个状态机在某组引脚上实现SPI等接口，另一个状态机可运行相同程序，映射至不同引脚组，从而提供第二个SPI接口。

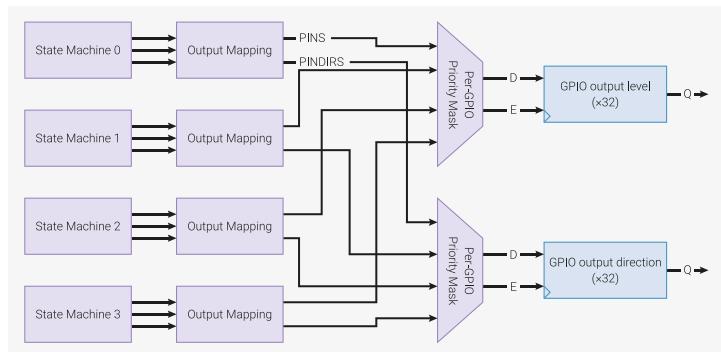
在任一时钟周期内，状态机可能执行 OUT 或 SET 操作，同时亦可执行侧置操作。

引脚映射逻辑为输出电平及输出使能寄存器生成32位写入掩码及写入数据总线，基于该请求及针脚映射配置。

如果旁路集（side-set）与同一状态机在同一周期执行的 OUT/SET 操作重叠，则旁路集在重叠区域内优先。

3.5.6.1. 输出优先级

图49。各状态机对每个GPIO写掩码的优先级选择。每个GPIO考虑四个状态机的电平和方向写入，并采用编号最高的状态机的值。



每个状态机可通过其针脚映射硬件在每个周期中断言一个 OUT/SET 信号和一个旁路集。这为每个状态机生成 GPIO 输出电平和输出使能寄存器的32位写数据及写掩码。

对于每个GPIO，PIO汇总来自所有四个状态机的写入信号，并采用编号最高的状态机的写入

内容。该过程分别针对输出电平和输出方向独立执行——同一周期内，状态机可同时更改同一引脚的电平和方向（例如通过同时的SET和旁路集），或由一个状态机更改GPIO方向，而另一个状态机更改该GPIO的电平。如果没有状态机声明写入GPIO的电平或方向，数值将保持不变。

每个状态机与引脚映射逻辑之间设有一级寄存器，输入映射逻辑与每个状态机之间亦设一级寄存器。假设传播延迟为零，状态机观察自身输出时存在如下延迟：

- 绕过同步器时，延迟为两个周期
- 启用同步器时，延迟为四个周期

3.5.6.2. 输入映射

IN指令观察到的数据映射为最低有效位对应由PINCTRL_IN_BASE选定的GPIO，且随后的更高有效位依次对应编号更高的GPIO，编号超过31后循环回绕。

换言之，IN总线即GPIO输入值按PINCTRL_IN_BASE所定义的值右旋转所得。若GPIO数量不足32个，PIO输入将以零补齐至32位。

某些指令，例如WAIT GPIO，使用绝对GPIO编号，而非对IN数据总线的索引。在这种情况下，右旋转操作不会被应用。

3.5.6.3. 输入同步器

为防止PIO出现亚稳态，每个GPIO输入均配备标准的两触发器同步器。这会增加两个周期的输入采样延迟，但其优点是状态机可在任意时刻执行IN PINS操作，并且仅会读取到干净的高或低电平，而非可能干扰状态机电路的中间状态。这对于诸如UART RX等异步接口而言是绝对必要的。

可以针对每个GPIO逐个选择绕过同步器。此举虽可减少输入延迟，但用户必须保证状态机不会在不适当的时间采样其输入信号。通常这仅适用于SPI等同步接口。通过设置INPUT_SYNC_BYPASS寄存器中的相应位即可绕过同步器。

● 警告

采样亚稳态输入可能导致状态机行为不可预测，应避免此类情况。除非应用于引脚的数据满足相对于CLK_SYS的建立时间和保持时间，否则不得禁用同步器。

3.5.7. 强制执行及 EXEC 指令

除了指令存储器外，状态机还可以从其他三个来源执行指令：

- MOV EXEC 执行来自寄存器 Source的指令
- OUT EXEC 执行从OSR移出的数据
- 系统可写入以供立即执行的SMx_INSTR控制寄存器

```

1 .program exec_example
2
3 hang:
4     jmp hang
5 execute:
6     out exec, 32
7     jmp execute
8

```

```

9 .program instructions_to_push
10
11     out x, 32
12     in x, 32
13     push

```

```

1 #include "tb.h" // TODO 本文件基于现有软件架构构建, 以支持 printf 等功能
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "exec_example.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     for (int i = 0; i < count_of(exec_example_program); ++i)
15         mm_pio->instr_mem[i] = exec_example_program[i];
16
17     // 启用自动拉取, 阈值设置为 32
18     mm_pio->sm[0].shiftctrl = (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB);
19
20     // 启动状态机 0 —— 将停留在 “hang” 循环
21     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
22
23     // 强制跳转至程序位置 1
24     mm_pio->sm[0].instr = 0x0000 | 0x1; // 执行 jmp 指令
25
26     // 向 FIFO 输入混合指令和数据
27     mm_pio->txf[0] = instructions_to_push_program[0]; // 输出 x, 32 位
28     mm_pio->txf[0] = 12345678; // 待 OUT 的数据
29     mm_pio->txf[0] = instructions_to_push_program[1]; // 在 x 寄存器中, 32
30     mm_pio->txf[0] = instructions_to_push_program[2]; // push
31
32     // 推入 TX FIFO 的程序将在 RX FIFO 返回部分数据
33     while (mm_pio->fstat & (1u << PIO_FSTAT_RXEMPTY_LSB))
34     ;
35
36     printf("%d\n", mm_pio->rx[0]);
37
38     return 0;
39 }

```

此处我们将示例程序加载入状态机，该程序执行以下两项操作：

- 进入无限循环
- 进入一个循环，反复从 TX FIFO 拉取 32 位数据，并将低 16 位作为指令执行

C 程序启动状态机，此时状态机进入 `hang` 循环状态。在状态机仍运行时，C 程序强制插入一个 `jmp` 指令，导致状态机跳出循环。

当指令写入 `INSTR` 寄存器时，状态机会立即解码并执行该指令，而非执行原本应从 PIO 指令存储器中取出的指令。程序计数器不会前进，因此在下一周期（假设写入 `INSTR` 接口的指令未阻塞），状态机会从暂停处继续执行当前程序，除非写入的指令本身

操控了 **PC**。

写入 **INSTR**寄存器的指令将忽略延迟周期，立即执行，并跳过状态机时钟分频器。此接口用于执行初始配置及控制流变更，因此无论状态机配置如何，都能及时执行指令。

写入 **INSTR**寄存器的指令允许发生阻塞，状态机会在内部锁存该指令直至完成。此状态由**EXECCTRL_EXEC_STALLED**标志指示。该标志可通过重启状态机或向**INSTR**写入 **NOP**指令予以清除。

在示例状态机程序的第二阶段，使用**OUT EXEC**指令。**OUT**本身占用一个执行周期，而 **OUT**所执行的指令位于下一个执行周期。请注意，我们执行的指令之一也是一个 **OUT** —— 状态机在任何周期内仅能执行一个 **OUT**指令。

OUT EXEC通过将 **OUT**移位数据写入内部指令锁存器来实现。在下一个周期，状态机会记住必须从该锁存器而非指令存储器执行，并且知晓该第二周期内不应递增 **PC**。

该程序运行时将打印"12345678"。

⚠ 注意

若写入 **INSTR**的指令阻塞，则其被存储于由**OUT EXEC**和**MOV EXEC**共用的指令锁存器中，且会覆盖锁存器内正在执行的指令。若使用 **EXEC**指令，写入 **INSTR**的指令不得阻塞。

3.6. 示例

这些示例通过实现常见的输入输出接口，展示了PIO的部分硬件特性。

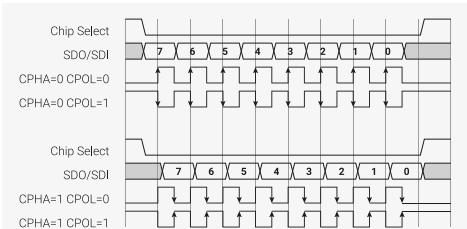
准备开始吗？

《Raspberry Pi Pico系列C/C++ SDK》一书包含了详尽的PIO章节，逐步讲解如何编写和构建首个PIO应用程序，并对部分程序逐行解析。该章节还涉及使用PIO与DMA的更广泛主题，并深入探讨了PIO

如何集成到您的软件中。

3.6.1. 双工 SPI

图50。在SPI通信中，主机和设备通过一对双向串行数据线交换数据，数据传输与时钟（SCK）同步进行。两个标志位CPOL和CPHA用于定义时钟的工作。CPOL表示时钟的空闲状态：0为低电平，1为高电平。时钟会发出多个脉冲，每个脉冲在两个方向传输一位数据，但始终回归空闲状态。CPHA决定在时钟的哪个边沿采样数据：0表示前沿，1表示后沿。图中箭头表示主机和设备均在该时钟边沿捕获数据。



SPI是一种常见的串行接口，其发展历史较为复杂。以下程序实现了全双工（即同时双向传输数据）的SPI，CPHA参数设为0。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第14至32行

```

14 .program spi_cpha0
15 .side_set 1
16
17 ; 引脚分配：
18 ; - SCK 为 side-set 引脚 0
19 ; - MOSI 为 OUT 引脚 0
20 ; - MISO 为 IN 引脚 0
21 ;
22 ; 必须启用自动推送和自动拉取，串行帧大小通过
23 ; 配置推送/拉取阈值来设定。左/右移位操作可用，但需
24 ; 自行调整数据对齐。这对于帧大小为
25 ; 通过使用窄存储复制和窄加载字节实现25; 8位或16位
26 ; RP2040的IO结构的选取行为。
27
28 ; 时钟相位 = 0: 数据在每个SCK脉冲的上升沿采集，且
29 ; 在下降沿发生转换，或在第一个上升沿之前某个时间点发生转换。
30
31     输出引脚, 1侧0 [1]; 此处空时停顿（即使侧设继续执行,
32     输入引脚, 1侧1 [1]; 指令停顿，因此我们在SCK低电平时停顿)

```

该代码使用autopush和autopull持续从FIFO流式传输数据。整个程序针对传输的每个位执行一次，然后循环执行。状态机跟踪已移入/移出的位数，并在适当时机自动推入/拉出FIFO。类似程序用于处理CPHA=1的情况：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第34至42行

```

34 .program spi_cpha1
35 .side_set 1
36
37 ; 时钟相位 = 1: 数据在每个 SCK 脉冲的上升沿发生变化,
38 ; 并在下降沿被捕获。
39
40     out x, 1      side 0      ; 空时暂停（保持 SCK 处于非激活状态）
41     mov pins, x side 1 [1] ; 输出数据，同时置位 SCK (mov pins 使用 OUT 映射)
42     in pins, 1    side 0      ; 输入数据，复位 SCK

```

注意

这些程序不控制片选线；由于不同 SPI 硬件行为差异显著，片选通常由软件控制的 GPIO 实现。上述链接中的完整 `spi.pio` 源代码包含了 PIO 实现硬件片选线的示例。

C 语言辅助函数用于配置状态机、连接 GPIO 并启动状态机运行。请注意，SPI 帧大小——即每个 FIFO 记录传输的位数——可编程为 1 至 32 之间的任意值，无需修改程序。配置完成后，状态机即开始运行。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第46至72行

```

46 static inline void pio_spi_init(PIO pio, uint sm, uint prog_offs, uint n_bits,
47     float clkdiv, bool cpha, bool cpol, uint pin_sck, uint pin_mosi, uint pin_miso) {
48     pio_sm_config c = cpha ? spi_cpha1_program_get_default_config(prog_offs) :
49         spi_cpha0_program_get_default_config(prog_offs);
50     sm_config_set_out_pins(&c, pin_mosi, 1);
51     sm_config_set_in_pins(&c, pin_miso);
52     sm_config_set_sideset_pins(&c, pin_sck);
53     // 本示例代码仅支持最高有效位优先（左移，自动推送/拉取，阈值=nbits）

54     sm_config_set_out_shift(&c, false, true, n_bits);
55     sm_config_set_in_shift(&c, false, true, n_bits);
56     sm_config_set_clkdiv(&c, clkdiv);

57     // MOSI、SCK输出为低电平，MISO为输入
58     pio_sm_set_pins_with_mask(pio, sm, 0, (1u << pin_sck) | (1u << pin_mosi));
59     pio_sm_set_pindirs_with_mask(pio, sm, (1u << pin_sck) | (1u << pin_mosi), (1u << pin_sck)
60     | (1u << pin_mosi) | (1u << pin_miso));
61     pio_gpio_init(pio, pin_mosi);
62     pio_gpio_init(pio, pin_miso);
63     pio_gpio_init(pio, pin_sck);

64     // 引脚多路复用器可被配置为反转输出（以及其他功能）
65     // 这是一种使CPOL=1 的简便方法
66     gpio_set_outover(pin_sck, cpol ? GPIO_OVERRIDE_INVERT : GPIO_OVERRIDE_NORMAL);
67     // SPI 是同步的，因此绕过输入同步器以降低输入延迟。
68     hw_set_bits(&pio->input_sync_bypass, 1u << pin_miso);
69
70     pio_sm_init(pio, sm, prog_offs, &c);
71     pio_sm_set_enabled(pio, sm, true);
72 }
```

状态机将立即开始输出 TX FIFO 中的任何数据，并将接收的数据推入 RX FIFO。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/pio_spi.c 第18至34行

```

18 void __time_critical_func(pio_spi_write8_blocking)(const pio_spi_inst_t *spi, const uint8_t
    *src, size_t len) {
19     size_t tx_remain = len, rx_remain = len;
20     // 对 FIFO 进行 8 位访问，使写入数据按字节复制。
21     // 该操作可免费实现左对齐（针对最高有效位优先移位输出）
22     io_rw_8 *txfifo = (io_rw_8 *) &spi->pio->txf[spi->sm];
23     io_rw_8 *rxfifo = (io_rw_8 *) &spi->pio->rxf[spi->sm];
24     while (tx_remain || rx_remain) {
25         if (tx_remain && !pio_sm_is_tx_fifo_full(spi->pio, spi->sm)) {
26             *txfifo = *src++;
27             --tx_remain;
28         }
29     }
30 }
```

```

29         if (rx_remain && !pio_sm_is_rx_fifo_empty(spi->pio, spi->sm)) {
30             (void) *rxfifo;
31             --rx_remain;
32         }
33     }
34 }
```

综合以上内容，该完整的C程序将通过一个1MHz的PIO SPI进行数据回环，覆盖所有四种CPOL/CPHA组合：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi_loopback.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4  * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "pio_spi.h"
12
13 // 本程序实例化了每种可能的
14 // CPOL/CPHA组合的PIO SPI，串行输入和输出引脚映射为
15 // 相同的GPIO。写入状态机TX FIFO的任何数据
16 // 应被串行化、反序列化，并重新出现在状态机的RX FIFO中。
17
18 #define PIN_SCK 18
19 #define PIN_MOSI 16
20 #define PIN_MISO 16 // 与MOSI相同，实现回环
21
22 #define BUF_SIZE 20
23
24 void test(const pio_spi_inst_t *spi) {
25     static uint8_t txbuf[BUF_SIZE];
26     static uint8_t rdbuf[BUF_SIZE];
27     printf("TX:");
28     for (int i = 0; i < BUF_SIZE; ++i) {
29         txbuf[i] = rand() >> 16;
30         rdbuf[i] = 0;
31         printf(" %02x", (int)txbuf[i]);
32     }
33     printf("\n");
34
35     pio_spi_write8_read8_blocking(spi, txbuf, rdbuf, BUF_SIZE);
36
37     printf("RX:");
38     bool mismatch = false;
39     for (int i = 0; i < BUF_SIZE; ++i) {
40         printf(" %02x", (int)rdbuf[i]);
41         mismatch = mismatch || rdbuf[i] != txbuf[i];
42     }
43     if (mismatch)
44         printf("\n不匹配\n");
45     else
46         printf("\n成功\n");
47 }
48
49 int main() {
50     stdio_init_all();
```

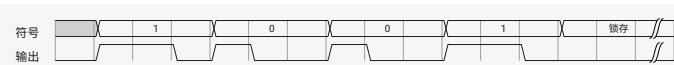
```

51
52     pio_spi_inst_t spi = {
53         .pio = pio0,
54         .sm = 0
55     };
56     float clkdiv = 31.25f; // 1 MHz @ 125 clk_sys
57     uint cpha0_prog_offs = pio_add_program(spi.pio, &spi_cpha0_program);
58     uint cpha1_prog_offs = pio_add_program(spi.pio, &spi_cpha1_program);
59
60     for (int cpha = 0; cpha <= 1; ++cpha) {
61         for (int cpol = 0; cpol <= 1; ++cpol) {
62             printf("CPHA = %d, CPOL = %d\n", cpha, cpol);
63             pio_spi_init(spi.pio, spi.sm,
64                         cpha ? cpha1_prog_offs : cpha0_prog_offs,
65                         8,           // 每个 SPI 帧 8 位
66                         clkdiv,
67                         cpha,
68                         cpol,
69                         PIN_SCK,
70                         PIN_MOSI,
71                         PIN_MISO
72                     );
73             test(&spi);
74             sleep_ms(10);
75         }
76     }
77 }
```

3.6.2. WS2812 LED

WS2812 LED 由专有脉宽串行格式驱动，宽正脉冲表示“1”位，窄正脉冲表示“0”。每个 LED 拥有一个串行输入和一个串行输出；LED 以链式连接，每个串行输入连接到前一 LED 的串行输出。

图 51. WS2812 线路格式。宽正脉冲表示1，窄正脉冲表示0，极长负脉冲表示锁存使能



LED 消耗 24 位像素数据，随后将任何额外输入数据传递到其输出。通过此方式，单个串行数据流可单独编程链中每个 LED 的颜色。一个较长的负脉冲将像素数据锁存至 LED。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> 第8行至第31行

```

8 .program ws2812
9 .side_set 1
10
11 ; 以下常量被选以确保与WS2812、
12 ; WS2812B及SK6812 LED的广泛兼容。其他常量可能支持
13 ; 特定LED的更高带宽，如WS2812B LED所用的(7,10,8)常量。
14
15 .define public T1 3
16 .define public T2 3
17 .define public T3 4
18
19 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
20 .lang_opt python out_init      = pico.PIO.OUT_HIGH
21 .lang_opt python out_shiftdir = 1
22
23 .wrap_target
```

```

24 bitloop:
25     out x, 1           side 0 [T3 - 1] ;当指令停顿时，侧边设定仍会发生
26     jmp !x do_zero side 1 [T1 - 1] ;根据移出位进行跳转。正脉冲
27 do_one:
28     jmp bitloop    side 1 [T2 - 1] ;继续保持高电平，形成长脉冲
29 do_zero:
30     nop            side 0 [T2 - 1] ;或保持低电平，形成短脉冲
31 .wrap

```

该程序将OSR中的位移入X寄存器，并根据每个位数据值，在侧边设定引脚0上产生宽脉冲或窄脉冲。必须配置自动拉取（`autopull`），阈值设为24。软件随后可向FIFO写入24位像素值，数据将被串行发送至一串WS2812 LED。该 `.pio` 文件包含用于配置此功能的C语言辅助函数：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> 第36行至52行

```

36 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
37                                         bool rgbw) {
38     pio_gpio_init(pio, pin);
39     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
40
41     pio_sm_config c = ws2812_program_get_default_config(offset);
42     sm_config_set_sideset_pins(&c, pin);
43     sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
44     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
45
46     int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
47     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
48     sm_config_set_clkdiv(&c, div);
49
50     pio_sm_init(pio, sm, offset, &c);
51     pio_sm_set_enabled(pio, sm, true);
52 }

```

由于移位按最高有效位优先进行，且我们的像素尺寸非二的幂次（因此无法依赖RP2040上的窄写复制行为来展开位），故需对写入TX FIFO的值进行预移位。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c> 第43至45行

```

43 static inline void put_pixel(PIO pio, uint sm, uint32_t pixel_grb) {
44     pio_sm_put_blocking(pio, sm, pixel_grb << 8u);
45 }

```

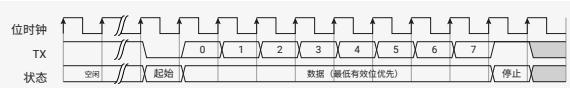
若采用DMA传输像素，则可将`autopull`阈值设为8位，DMA传输单位设为8位，并每次向FIFO写入一个字节。每个像素将由3次单字节传输组成。由于RP2040上的总线结构和DMA的工作原理，DMA传输的每个字节在写入32位IO寄存器时会被复制四次，因此数据实际上位于移位寄存器的两端，您可以放心地向任意方向进行移位。

需要更详细的信息吗？

WS2812示例是[Raspberry Pi Pico系列C/C++ SDK](#)文档中PIO章节的教程内容。该教程逐行解析 `ws2812` 程序，追踪程序的执行过程，并展示程序各阶段GPIO输出的波形图。

3.6.3. UART 发送

图52. UART串行格式。线路空闲时电平为高。发送端将线路拉低一个比特周期以标志串行帧开始（“起始位”），随后传输固定数量的数据位。线路在下一个串行帧开始之前至少保持一个比特周期（“停止位”）的空闲状态。



本程序实现了通用异步收发器（UART）串行外设的发送部分。或许可更准确地称其为UAT。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio，第8至18行

```
8 .program uart_tx
9 .side_set 1 opt
10
11 ; 一款8n1 UART发送程序。
12 ; OUT引脚0和side-set引脚0均映射至UART TX引脚。
13
14     pull      side 1 [7] ; 断言停止位，或保持线路处于空闲状态等待
15     set x, 7    side 0 [7] ; 预加载比特计数器，断言起始位持续8个时钟周期
16     bitloop:           ; 该循环将运行8次（8n1 UART）
17         输出引脚, 1          ; 将OSR中1位数据移位至第一个OUT引脚
18         jmp x-- bitloop    [6] ; 每次循环迭代包含8个周期。
```

按当前代码，该程序将：

- 使引脚保持高电平阻塞，直到数据出现（注意即使状态机阻塞，侧置功能仍然生效）
- 断言起始位，持续8个状态机执行周期
- 移位输出8位数据，每位持续8个周期
- 返回空闲线路状态，至少维持8个周期，随后断言下一起始位

若状态机的时钟分频器配置为以目标波特率的8倍运行，则该程序会在数据由软件或系统DMA推送至TX FIFO时，传输格式规范的UART串行帧。若需扩展以支持不同帧长（不同数据位数），可将`set x, 7`替换为`mov x, y`，使`y`寄存器成为每个状态机的UART帧大小配置寄存器。

SDK中的.pio文件也包含此功能，用于配置引脚和状态机，该程序一旦加载至PIO指令存储器：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第24至51行

```
24 static inline void uart_tx_program_init(PIO pio, uint sm, uint offset, uint pin_tx, uint
baud) {
25     // 指示PIO在所选引脚上初始输出高电平，然后通过IO复用器将PIO映射至该引脚。
26     // onto that pin with the IO muxes.
27     pio_sm_set_pins_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
28     pio_sm_set_pindirs_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
29     pio_gpio_init(pio, pin_tx);
30
31     pio_sm_config c = uart_tx_program_get_default_config(offset);
32
33     // 输出向右移动，无自动拉取
34     sm_config_set_out_shift(&c, true, false, 32);
35
36     // 我们将OUT和side-set映射到同一引脚，因为有时
37     // 需要将用户数据（通过OUT）输出到引脚，有时
38     // 需要输出固定值（起始位/停止位）
39     sm_config_set_out_pins(&c, pin_tx, 1);
40     sm_config_set_sideset_pins(&c, pin_tx);
```

```

41
42 // 我们仅需要TX，因此配置一个深度为8的FIFO！
43 sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
44
45 // 状态机每8个执行周期传输1位。
46 float div = (float)clock_get_hz(clk_sys) / (8 * baud);
47 sm_config_set_clkdiv(&c, div);
48
49 pio_sm_init(pio, sm, offset, &c);
50 pio_sm_set_enabled(pio, sm, true);
51 }

```

状态机配置为在 `out` 指令中执行右移，因为UART通常按最低有效位（LSB）优先发送数据。配置完成后，状态机将输出推入TX FIFO的所有字符。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第53至55行

```

53 static inline void uart_tx_program_putc(PIO pio, uint sm, char c) {
54     pio_sm_put_blocking(pio, sm, (uint32_t)c);
55 }

```

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第57至60行

```

57 static inline void uart_tx_program_puts(PIO pio, uint sm, const char *s) {
58     while (*s)
59         uart_tx_program_putc(pio, sm, *s++);
60 }

```

SDK中的示例程序将配置一个PIO状态机作为UART TX外设，并以115200波特率每秒在GPIO 0上打印消息。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.c

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符：BSD-3-Clause
5 */
6
7 #include "pico/stdlib.h"
8 #include "hardware/pio.h"
9 #include "uart_tx.pio.h"
10
11 // 我们将使用PIO在同一GPIO上打印“Hello, world!”，该GPIO通常连接UART0。
12 // 通常将UART0附接至该GPIO。
13 #define PIO_TX_PIN 0
14
15 // 检查引脚是否与平台兼容
16 #error 尝试在不支持的平台上使用引脚编号>=32
17
18 int main() {
19     // 此波特率与Pico上UART的默认波特率相同
20     const uint SERIAL_BAUD = 115200;
21
22     PIO pio;
23     uint sm;
24     uint offset;
25

```

```

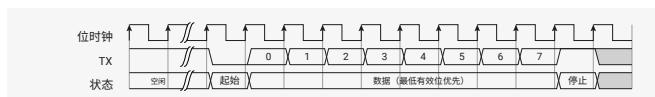
26 // 这将为我们的程序寻找一个空闲的 PIO 和状态机，并加载到程序中
27 // 我们使用 pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range 变体)
28 // 因此，如果需要且硬件支持，将获得一个适用于访问 gpio ≥ 32 的 PIO 实例

29     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_tx_program, &pio,
30     &sm, &offset, PIO_TX_PIN, 1, true);
31     hard_assert(success);
32
33     uart_tx_program_init(pio, sm, offset, PIO_TX_PIN, SERIAL_BAUD);
34
35     while (true) {
36         uart_tx_program_puts(pio, sm, "Hello, world! (from PIO!)\r\n");
37         sleep_ms(1000);
38     }
39
40     // 这将释放资源并卸载我们的程序
41     pio_remove_program_and_unclaim_sm(&uart_tx_program, pio, sm, offset);
42 }
```

RP2040上的两个PIO实例可扩展为8个额外的UART TX接口，分布在8个不同引脚，支持8种不同波特率。

3.6.4. UART 接收

回顾图52，展示了8n1 UART的格式：



通过等待起始位、以正确时序采样8次并将结果推送至RX FIFO，我们可以恢复数据。以下极可能是实现该功能的最简程序：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第8至19行

```

8 .program uart_rx_mini
9
10 ; 最简可用的8n1 UART接收器。等待起始位，然后采样8位
11; 确保正确时序。
12 ; IN引脚0映射至用于UART RX的GPIO。
13 ; 必须启用自动推送，阈值设置为8。
14
15     wait 0 pin 0          ; 等待起始位
16     set x, 7 [10]         ; 预加载位计数器，延迟至第一个数据位的采样点
17     bitloop:              ; 循环8次
18     in pins, 1             ; 采样数据
19     jmp x-- bitloop [6]   ; 每次迭代为8个周期
```

该方法可行，但存在一些不便之处，例如当线路持续低电平时，会重复输出 **NUL** 字符。

理想情况下，我们应丢弃未被起始位和停止位正确框定的数据（并设置一个粘性标志以指示该情况），且当线路长时间持续低电平时暂停接收。我们可以将这些功能添加至程序，代价是增加若干指令。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第44至63行

```

44 .program uart_rx
45
46 ; 微稍更完善的8n1 UART接收器，可更稳妥地处理帧错误和
```

```

47 ;断帧条件。
48 ;IN引脚0和JMP引脚均映射为用作UART RX的GPIO。
49
50 start:
51     等待引脚0为0          ;阻塞，直至检测到起始位
52     设定x为7 [10]；预加载位计数器，随后延时至第一个数据位中点
53 bitloop:           ;（含等待及设定，计12个周期）。
54     从引脚读取，1        ;将数据位移入ISR
55     jmp x--_bitloop [6] ;循环8次，每次循环耗时8周期
56     jmp pin good_stop  ;检测停止位（应为高电平）
57
58     irq 4 rel          ;出现帧错误或断帧时，设置一个粘性标志，
59     等待 1 针 0          ;并等待线路返回空闲状态。
60     jmp start           ;未检测到有效帧时，请勿发送数据。
61
62 good_stop:         ;返回起始点前不延迟；这在
63     发送               ;TX时钟稍快时尤为重要。

```

第二个示例不使用自动发送（第3.5.4节），而是采用显式的 `push` 指令，以便根据是否检测到正确停止位来决定是否发送。该 `.pio` 文件包含一个辅助函数，用以配置状态机并连接启用上拉的 GPIO：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第67至85行

```

67 static inline void uart_rx_program_init(PIO pio, uint sm, uint offset, uint pin, uint baud) {
68     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
69     pio_gpio_init(pio, pin);
70     gpio_pull_up(pin);
71
72     pio_sm_config c = uart_rx_program_get_default_config(offset);
73     sm_config_set_in_pins(&c, pin); // 用于 WAIT 和 IN
74     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
75     // 右移，自动推送功能已禁用
76     sm_config_set_in_shift(&c, true, false, 32);
77     // 由于未进行任何 TX，使用更深的 FIFO
78     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
79     // 状态机每 8 个执行周期传输 1 位
80     float div = (float)clock_get_hz(clk_sys) / (8 * baud);
81     sm_config_set_clkdiv(&c, div);
82
83     pio_sm_init(pio, sm, offset, &c);
84     pio_sm_set_enabled(pio, sm, true);
85 }

```

为正确接收最低有效位优先发送的数据，ISR 配置为右移模式。移入 8 位后，遗憾的是 8 位数据位位于 ISR 的 31:24 位，最低有效位为 24 个零。这里的一个选项是使用一条 `null_24` 指令，将 ISR 内容下移至 7:0 位。另一种方法是从 FIFO 的偏移量 3 字节处进行 8 位读取，以使处理器总线硬件（或 DMA）能够自动选择相关字节：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第87至93行

```

87 static inline char uart_rx_program_getc(PIO pio, uint sm) {
88     // 从 FIFO 最高字节进行 8 位读取，数据左对齐
89     io_rw_8 *rxfifo_shift = (io_rw_8*)&pio->rxf[sm] + 3;
90     while (pio_sm_is_rx_fifo_empty(pio, sm))
91         tight_loop_contents();
92     return (char)*rxfifo_shift;
93 }

```

示例程序演示了如何使用该UART RX程序接收RP2040上的硬件UART所发送的字符。程序需将GPIO4连接至GPIO3方可正常运行。为了简化对3个不同串口的管理，本程序使用核心1在测试UART（UART 1）上输出字符串，而核心0运行的代码将从PIO状态机获取字符，并传递给用于调试控制台的UART（UART 0）。另一种方法是基于中断的IO，利用PIO的FIFO中断请求（IRQ）。若IRQ0_INTERRUPT寄存器中的SM0_RXNEEMPTY位被置位，则当状态机0的RX FIFO中有字符时，PIO将触发其第一个中断请求线。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.c

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stdlib.h"
10 #include "pico/multicore.h"
11 #include "hardware/pio.h"
12 #include "hardware/uart.h"
13 #include "uart_rx.pio.h"
14
15 // 本程序
16 // - 使用 UART1 (默认备用 UART) 发送文本
17 // - 使用 PIO 状态机接收该文本
18 // - 将接收到的文本打印至默认控制台 (UART0)
19 // 在某些开发板上, UART1 可能为默认 UART, 可能需重新配置。
20 // default UART.
21
22 #define SERIAL_BAUD PICO_DEFAULT_UART_BAUD_RATE
23 #define HARD_UART_INST uart1
24
25 // 需将 GPIO4 连接至 GPIO3
26 #define HARD_UART_TX_PIN 4
27 #define PIO_RX_PIN 3
28
29 // 请确认该引脚与平台兼容
30 #error 试图使用平台不支持的32号及以上引脚
31
32 // 请求核心1打印字符串, 以简化核心0的操作
33 void core1_main() {
34     const char *s = (const char *) multicore_fifo_pop_blocking();
35     uart_puts(HARD_UART_INST, s);
36 }
37
38 int main() {
39     // 控制台输出 (同样是 UART, 确实容易混淆)
40     setup_default_uart();
41     printf("Starting PIO UART RX example\n");
42
43     // 配置我们将用于打印字符的硬件 UART
44     uart_init(HARD_UART_INST, SERIAL_BAUD);
45     gpio_set_function(HARD_UART_TX_PIN, GPIO_FUNC_UART);
46
47     // 配置我们将用于接收字符的状态机
48     PIO pio;
49     uint sm;
50     uint offset;
51
52     // 该操作将为程序查找并占用空闲的 pio 和状态机, 并进行加载

```

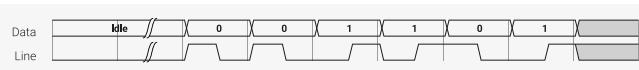
```

53 // 我们使用 pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range 变体)
54 // 因此，如果需要且硬件支持，将获得一个适用于访问 gpio ≥ 32 的 PIO 实例

55     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_rx_program, &pio,
56     &sm, &offset, PIO_RX_PIN, 1, true);
57     hard_assert(success);
58
59     uart_rx_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
60     //uart_rx_mini_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
61
62     // 指示核心1尽快将文本打印到uart1
63     multicore_launch_core1(core1_main);
64
65     multicore_fifo_push_blocking((uint32_t) text);
66
67     // 将从PIO接收的字符回显至控制台
68     while (true) {
69         char c = uart_rx_program_getc(pio, sm);
70         putchar(c);
71     }
72
73     // 此操作将释放资源并卸载程序
74     pio_remove_program_and_unclaim_sm(&uart_rx_program, pio, sm, offset);
75 }
```

3.6.5. 曼彻斯特编码串行发送与接收

图53. 曼彻斯特码串行线路编码。每个数据位由高脉冲后接低脉冲（表示“0”位）或低脉冲后接高脉冲（表示“1”位）表示。



Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第8至30行

```

8 .program manchester_tx
9 .side_set 1 opt
10
11 ; 每12个周期传输一位。“0”编码为高-低序列
12 ; （每部分持续半个比特周期，即6个周期），“1”编码为
13 ; 低-高序列。
14 ;
15 ; Side-set第0位必须映射至用于TX的GPIO。
16 ; 必须启用自动拉取——本程序不考虑阈值。
17 ; 程序从公共标签“start”处开始。
18
19 .wrap_target
20 do_1:
21     nop           side 0 [5] ; 低电平持续6个周期（包含5个延时周期，外加1个nop周期）
22     jmp get_bit side 1 [3] ; 高电平持续4个周期。'get_bit' 指令额外占用2个周期
23 do_0:
24     nop           side 1 [5] ; 输出高电平持续6个周期
25     nop           side 0 [3] ; 输出低电平持续4个周期
26 public start:
27 get_bit:
28     out x, 1           ; 始终将一个位从OSR移位至X寄存器，以便我们
29     jmp !x do_0          ; 根据该位进行分支。Autopull功能在OSR为空时会自动重新填充。
30 .wrap
```

从名为 `start` 的标签开始，本程序每次将一个数据位移入 X 寄存器，以便根据该位进行跳转。

该值。根据结果，使用 side-set 在所选 GPIO 上驱动 1-0 或 0-1 序列。该程序采用 autopull（第 3.5.4.2 节）功能，当移出一定量数据后，自动从 TX FIFO 补充 OSR，且不会中断程序控制流或时序。该功能由 .pio 文件中的辅助函数实现，该函数负责配置并启动状态机：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio，第 33 至 46 行

```

33 static inline void manchester_tx_program_init(PIO pio, uint sm, uint offset, uint pin, float
      div) {
34     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
35     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
36     pio_gpio_init(pio, pin);
37
38     pio_sm_config c = manchester_tx_program_get_default_config(offset);
39     sm_config_set_sideset_pins(&c, pin);
40     sm_config_set_out_shift(&c, true, true, 32);
41     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
42     sm_config_set_clkdiv(&c, div);
43     pio_sm_init(pio, sm, offset + manchester_tx_offset_start, &c);
44
45     pio_sm_set_enabled(pio, sm, true);
46 }
```

可编程另一个状态机以从传输信号中恢复原始数据：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第49至71行

```

49 .program manchester_rx
50
51 ; 假设线路空闲为低电平，第一位为0
52 ; 一位占12个周期
53 ; “0” 编码为 10
54 ; “1” 编码为 01
55 ;
56 ; IN 基址和 JMP 引脚映射必须指向用于接收的 GPIO。
57 ; 必须启用自动推送 (Autopush)。
58 ; 启用状态机前，应将其置于 “wait 1, pin” 状态，以便
59 ; 在初始线路空闲状态结束之前，采样不会开始。
60
61 start_of_0:           ; 我们已进入0信号的0.25位 — 信号为高电平
62     等待引脚0 变为0       ; 等待1→0跳变 — 此时已进入比特的0.5位
63     输入 y, 1 [8]         ; 发出0，休眠3/4位时长
64     jmp pin start_of_0 ; 若信号再次为1，则为另一个0位，否则为1
65
66 .wrap_target
67 start_of_1:           ; 我们已进入1信号的0.25位 — 信号为1
68     等待引脚1 变为0       ; 等待0→1跳变 — 此时已进入比特的0.5位
69     输入 x, 1 [8]         ; 发出1，休眠3/4位时长
70     jmp pin start_of_0 ; 若信号再次为0，则为另一个1位，否则为0
71 .wrap
```

此处的主要难点在于保持对输入转换的同步，因为发射器和接收器的时钟可能会相互漂移。在曼彻斯特码中，符号中间始终存在一次状态转换，且基于初始线路状态（高电平或低电平），我们可确定该转换的方向，因此可利用 wait 指令在每个数据位上重新同步线路转换。

本程序要求 X 和 Y 寄存器分别初始化为 1 和 0，以便向 in 指令提供恒定的 1 或 0。配置状态机的代码通过执行若干 set 指令初始化这些寄存器，随后启动程序运行。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第74至94行

```

74 static inline void manchester_rx_program_init(PIO pio, uint sm, uint offset, uint pin, float
    div) {
75     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
76     pio_gpio_init(pio, pin);
77
78     pio_sm_config c = manchester_rx_program_get_default_config(offset);
79     sm_config_set_in_pins(&c, pin); // 用于 WAIT
80     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
81     sm_config_set_in_shift(&c, true, true, 32);
82     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
83     sm_config_set_clkdiv(&c, div);
84     pio_sm_init(pio, sm, offset, &c);
85
86     // 将 X 和 Y 分别设置为 0 和 1，以便方便地发送到 ISR/FIFO。
87     pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
88     pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
89     // 假设线路空闲时为低电平，且首个发送的位为 0。启用前将状态机置于
90     // 等待状态。RX 将在检测到第一个 0 符号后启动。
91     // detected.
92     pio_sm_exec(pio, sm, pio_encode_wait_pin(1, 0) | pio_encode_delay(2));
93     pio_sm_set_enabled(pio, sm, true);
94 }
```

SDK中的示例C程序将以约10Mbps的速度从GPIO2向GPIO3传输曼彻斯特码串行数据
(假设系统时钟频率为125MHz)。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.c 第20至43行

```

20 int main() {
21     stdio_init_all();
22
23     PIO pio = pio0;
24     uint sm_tx = 0;
25     uint sm_rx = 1;
26
27     uint offset_tx = pio_add_program(pio, &manchester_tx_program);
28     uint offset_rx = pio_add_program(pio, &manchester_rx_program);
29     printf("传输程序加载于%d\n", offset_tx);
30     printf("接收程序加载于%d\n", offset_rx);
31
32     manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 1.f);
33     manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 1.f);
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }
```

3.6.6. 差分曼彻斯特编码（BMC）发送与接收

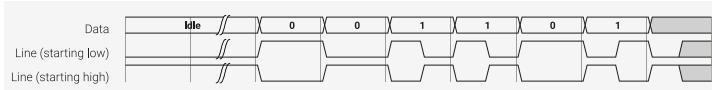
图54。差分曼彻斯特特码，又称双相标记码（BMC）。线路在每个位周期开始时发生跳变。

位周期中间的

跳变表示数据位 1

，

以及缺失，即一个 0 比特。这些编码规则无论线路的初始状态是高电平还是低电平均相



发送程序类似于曼彻斯特编码示例：它不断地将一个比特从OSR移位至X（依赖自动拉取机制在后台重新填充OSR），根据该比特的值分支，并驱动GPIO引脚的电平升降。
增加的复杂性在于，我们驱动至引脚的信号模式不仅取决于数据比特的值（如传统曼彻斯特编码），还取决于上一个比特周期结束时线路所保持的电平状态。如图54所示，当线路初始为高电平时，信号模式将被反转。为应对这一情况，存在两份测试与驱动代码，分别对应两种初始线路状态，并通过一系列跳转指令以正确顺序将它们连接起来。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 8-35行

```

8 .program differential_manchester_tx
9 .side_set 1 opt
10
11 ; 每16个周期传输1个位。在每个位周期内：
12 ; - '0' 编码为位周期开始时的跳变
13 ; - '1' 编码为位周期开始和中间均有跳变
14 ;
15 ; side-set位0必须映射至数据输出引脚。
16 ; 必须启用autopull功能。
17
18 public start:
19 initial_high:
20     out x, 1           ; 位周期开始时：始终产生跳变
21     jmp !x high_0     side 1 [6] ; 测试刚从OSR移出的数据位
22 high_1:
23     nop
24     jmp initial_high side 0 [6] ; 对于 '1' 位，位周期中间也需跳变
25 high_0:
26     jmp initial_low      [7] ; 否则，该线路在中间保持稳定
27
28 initial_low:
29     out x, 1           ; 始终将1位从OSR移位至X，因此我们可以
30     jmp !x low_0        side 0 [6]; 据此分支。Autopull为我们自动补充OSR。
31 low_1:
32     nop
33     jmp initial_low    side 1 [6]; 如果有两次跳变，则返回
34 low_0:
35     jmp initial_high    [7] ; 初始线路状态已翻转!

```

该 .pio 文件还包含一个辅助函数，用于初始化差分曼彻斯特发送（TX）状态机，并将其连接至指定GPIO。我们任意选择了32位帧长和最低有效位优先的序列化（`shift_to_right`在`sm_config_set_out_shift`中设为true），但由于程序一次操作一位，我们可通过重新配置状态机予以修改。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第38至53行

```

38 static inline void differential_manchester_tx_program_init(PIO pio, uint sm, uint offset,
      uint pin, float div) {
39     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
40     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
41     pio_gpio_init(pio, pin);
42
43     pio_sm_config c = differential_manchester_tx_program_get_default_config(offset);
44     sm_config_set_sideset_pins(&c, pin);
45     sm_config_set_out_shift(&c, true, true, 32);
46     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
47     sm_config_set_clkdiv(&c, div);

```

```

48     pio_sm_init(pio, sm, offset + differential_manchester_tx_offset_start, &c);
49
50 // 执行阻塞拉取，以保持初始线路状态直到数据可用

51     pio_sm_exec(pio, sm, pio_encode_pull(false, true));
52     pio_sm_set_enabled(pio, sm, true);
53 }

```

接收程序采用以下策略：

- 等待直到位周期开始时的初始跳变，确保与发送时钟保持同步
- 然后等待配置的位周期的3/4，以使采样点位于半位周期的后半部分中心（参见图54）
- 在该点采样线路，以确定该比特周期内是否存在一次或两次跳变
- 重复

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第55至85行

```

55 .program differential_manchester_rx
56
57 ; 假定线路空闲时为低电平
58 ; 一个比特周期为 16 个周期。在每个比特周期内：
59 ; - “0”的编码为在时间 0 发生一次跳变
60 ; - “1”的编码为在时间 0 及时间 T/2 各发生一次跳变
61 ;
62 ; IN 映射和 JMP 引脚选择均须映射至所用 GPIO
63 ; RX 数据。必须启用自动推送 (Autopush)。
64
65 public start:
66 initial_high:           ; 在比特周期开始处检测上升沿
67     wait 1 pin, 0 [11] ; 延时至第二半周期的眼位 (即 3/4 处)
68     jmp pin high_0      ; 通过位) 并根据 RX 引脚的高/低电平进行跳转。
69 high_1:
70     in x, 1             ; 检测到第二次跳变 (`1` 数据符号)
71     jmp initial_high
72 high_0:
73     in y, 1 [1]         ; 线路仍为高电平，未检测到中间跳变 (数据为 `0`)
74     ; 直接落入
75
76 .wrap_target
77 initial_low:            ; 查找位周期起始处的下降沿
78     wait 0 pin, 0 [11] ; 延迟至半周期后半部分的采样时刻
79     jmp pin low_1
80 low_0:
81     in y, 1             ; 线路仍为低电平，未检测到中间跳变 (数据为 `0`)
82     jmp initial_high
83 low_1:                  ; 检测到第二次跳变 (数据为 `1`)
84     in x, 1 [1]
85 .wrap

```

该代码假设X和Y的值分别为1和0。此功能由所包含的C辅助函数实现：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第88至104行

```

88 static inline void differential_manchester_rx_program_init(PIO pio, uint sm, uint offset,
89               uint pin, float div) {
90     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
91     pio_gpio_init(pio, pin);
92

```

```

92     pio_sm_config c = differential_manchester_rx_program_get_default_config(offset);
93     sm_config_set_in_pins(&c, pin); // 用于 WAIT
94     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
95     sm_config_set_in_shift(&c, true, true, 32);
96     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
97     sm_config_set_clkdiv(&c, div);
98     pio_sm_init(pio, sm, offset, &c);
99
100    // X和Y被设定为0和1，便于向ISR/FIFO发送。
101    pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
102    pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
103    pio_sm_set_enabled(pio, sm, true);
104 }

```

所有组件均已具备，现可通过两路 GPIO 之间的一根导线实现串行数据的回环传输。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4  * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stlolib.h"
10 #include "hardware/pio.h"
11 #include "differential_manchester.pio.h"
12
13 // 差分串行收发示例
14 // 需将导线连接 GPIO2 至 GPIO3
15
16 const uint pin_tx = 2;
17 const uint pin_rx = 3;
18
19 int main() {
20     stdio_init_all();
21
22     PIO pio = pio0;
23     uint sm_tx = 0;
24     uint sm_rx = 1;
25
26     uint offset_tx = pio_add_program(pio, &differential_manchester_tx_program);
27     uint offset_rx = pio_add_program(pio, &differential_manchester_rx_program);
28     printf("发送程序加载于 %d\n", offset_tx);
29     printf("接收程序加载于 %d\n", offset_rx);
30
31     // 配置状态机，设置比特率为 5 Mbps
32     differential_manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 125.f / (16 * 5));
33     differential_manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 125.f / (16 * 5));
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }

```

3.6.7. I2C

图 55。一次 1 字节的 I2C 读取传输。
空闲状态下，两条线路均处于高电平漂浮状态。

主设备将 SDA 拉低（起始条件），随后发送 7 位地址 A6-A0 及一个方向位（读/写）。

目标设备将 SDA 拉低以确认地址（ACK），随后传输数据字节。目标设备沿 SDA 线串行传输数据，由 SCL 时钟控制输出。

每第 9 个时钟周期，主设备将 SDA 拉低以确认数据，最后一个字节除外，主设备保持线路为高电平（NAK 当 SCL 为高电平时释放 SDA 即为停止条件，令总线返回空闲状态。



I2C 是一种广泛使用的串行总线，最早记载于《死海古卷》，后由飞利浦半导体采用。两根线配合上拉电阻构成开漏总线，多个设备通过将总线拉低或释放线路让其被上拉到高电平，进行地址定位及信号通信。其具备若干特殊属性：

- SCL 信号线可由总线上的任何成员在任何时间、任意时长拉低（不必是传输的目标或发起方）。此现象称为时钟延长。在所有驱动器释放时钟之前，总线不会前进。
- 总线成员既可作为某次传输的目标，也可发起其他传输（主从角色不固定）。然而，大多数 I2C 硬件对此支持较差。
- SCL 不是边沿敏感的时钟线，而 SDA 必须在 SCL 处于高电平时始终有效。
- 尽管 SDA 对 SCL 具有透明性，但在 SCL 高电平期间 SDA 的跳变用于标记传输的开始和结束（起始/停止），或单次传输中的新地址阶段（重复启动）。

以下 PIO 程序在发起方角色中负责处理串行化、时钟延长及 ACK 检测。该程序提供一种机制，可在 FIFO 数据流中对 PIO 指令进行转义，以在适当时机发出起始/停止/重复启动序列。只要未收到意外的 NAK，该机制即可从 DMA 缓冲区执行长序列的 I2C 传输，无需处理器干预。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第 8 行至第 73 行

```

8 .program i2c
9 .side_set 1 opt pindirs
10
11 ; 发送编码：
12 ; | 15:10 | 9      | 8:1   | 0    |
13 ; | Instr | 结束 | 数据 | NAK |
14 ;
15 ; 若 Instr 的值 n > 0，则该 FIFO 字无
16 ; 数据负载，接下来的 n + 1 个字将作为指令执行。
17 ; 否则，先移出 8 位数据位，随后是 ACK 位。
18 ;
19 ; Instr 机制允许由处理器编程停止/启动/重复启动序列，
20 ; 并由状态机在数据流中的指定点执行，
21 ; in the datastream.
22 ;
23 ; "Final" 字段应在传输的最后一个字节中设置。
24 ; 此设置指示状态机忽略 NAK：若此字段未设置，
25 ; 则任何 NAK 将导致状态机停止并产生中断。
26 ;
27 ; 应启用自动拉取，阈值设为 16。
28 ; 应启用自动推送，阈值设为 8。
29 ; 应采用半字写入方式访问 TX FIFO，以确保
30 ; 数据能够即时在 OSR 中可用。
31 ;
32 ; 引脚映射：
33 ; - 输入引脚 0 为 SDA，1 为 SCL（若使用时钟拉伸）
34 ; - 跳转引脚为 SDA
35 ; - 侧设引脚 0 为 SCL
36 ; - 置位引脚 0 为 SDA
37 ; - 输出引脚 0 为 SDA
38 ; - SCL 必须为 SDA + 1（用于等待映射）
39 ;
40 ; 系统 IO 控制中 OE 输出应取反！
41 ; （该程序中可实现反转，
42 ; 但需消耗两条指令：一条用于反转，另一条用于相应处理）

```

```

43 ; 伴随对TX移位计数器执行MOV的副作用。
44
45 do_nack:
46     jmp y-- entry_point          ; 如果预期接收NAK，则继续
47     irq wait 0 rel              ; 否则停止，寻求协助
48
49 do_byte:
50     set x, 7                   ; 循环8次
51 bitloop:
52     out pindirs, 1             [7] ; 串行写入数据（读取时全为1）
53     nop                      side 1 [2]; SCL上升沿
54     wait 1 pin, 1              [4] ; 允许时钟拉伸
55     in pins, 1                [7] ; 在SCL脉冲中间采样读取数据
56     jmp x-- bitloop side 0 [7]; SCL下降沿
57
58 ; 处理ACK脉冲
59     out pindirs, 1             [7] ; 读取时，我们提供ACK。
60     nop                      side 1 [7]; SCL上升沿
61     wait 1 pin, 1              [7] ; 允许时钟拉伸
62     jmp pindir do_nack side 0 [2]; 测试SDA是否为ACK/NAK，若为ACK则继续执行
63
64 公共入口点：
65 .wrap_target
66     out x, 6                  ; 解包指令计数
67     out y, 1                  ; 解包NAK忽略位
68     jmp !x do_byte           ; 指令 == 0，此为数据记录。
69     out null, 32              ; 指令 > 0，此OSR剩余部分无效
70 do_exec:
71     out exec, 16              ; 每个FIFO字执行一条指令
72     jmp x-- do_exec          ; 重复n+1次
73 .wrap

```

I2C程序所需的IO映射较为复杂，原因在于两条串行线必须以不同方式进行驱动和采样。一个有趣的特性是，状态机必须在输出为低电平时将输出使能置高，因为总线为开漏结构，数据的逻辑含义被反转。该反转本可在PIO程序中处理（例如`mov osr, ~osr`），但也可利用RP2040的IO控制，在GPIO多路复用器中完成，从而节省一条指令。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第81至121行

```

81 static inline void i2c_program_init(PIO pio, uint sm, uint offset, uint pin_sda, uint
82     pin_scl) {
83     assert(pin_scl == pin_sda + 1);
84     pio_sm_config c = i2c_program_get_default_config(offset);
85
86     // IO映射
87     sm_config_set_out_pins(&c, pin_sda, 1);
88     sm_config_set_set_pins(&c, pin_sda, 1);
89     sm_config_set_in_pins(&c, pin_sda);
90     sm_config_set_sideset_pins(&c, pin_scl);
91     sm_config_set_jmp_pin(&c, pin_sda);
92
93     sm_config_set_out_shift(&c, false, true, 16);
94     sm_config_set_in_shift(&c, false, true, 8);
95
96     float div = (float)clock_get_hz(clk_sys) / (32 * 100000);
97     sm_config_set_clkdiv(&c, div);
98
99     // 尝试避免在连接IO时对总线引起毛刺。进行配置
100    // 使当PIO断言OE低电平时管脚被驱动为低电平，并且被上拉
101    // 否则。

```

```

101    gpio_pull_up(pin_scl);
102    gpio_pull_up(pin_sda);
103    uint32_t both_pins = (1u << pin_sda) | (1u << pin_scl);
104    pio_sm_set_pins_with_mask(pio, sm, both_pins, both_pins);
105    pio_sm_set_pindirs_with_mask(pio, sm, both_pins, both_pins);
106    pio_gpio_init(pio, pin_sda);
107    gpio_set_oeover(pin_sda, GPIO_OVERRIDE_INVERT);
108    pio_gpio_init(pio, pin_scl);
109    gpio_set_oeover(pin_scl, GPIO_OVERRIDE_INVERT);
110    pio_sm_set_pins_with_mask(pio, sm, 0, both_pins);
111
112    // 启动前清除IRQ标志，并确保该标志不会实际触发
113    // 系统级中断（我们将其用作状态标志）
114    pio_set_irq0_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
115        sm), false);
115    pio_set_irq1_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
116        sm), false);
116    pio_interrupt_clear(pio, sm);
117
118    // 配置并启动SM
119    pio_sm_init(pio, sm, offset + i2c_offset_entry_point, &c);
120    pio_sm_set_enabled(pio, sm, true);
121 }

```

我们还可以使用 PIO 汇编器生成一张指令表，用于通过 FIFO 传递，适用于启动/停止/重新启动条件。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第126至136行

```

126 .program set_scl_sda
127 .side_set 1 opt
128
129 ; 汇编一张指令表，软件可从中选择并传入 FIFO，以发出 START/STOP/RSTART 指令。此表并非设计
130   为完整程序运行。
131
132
133     set pindirs, 0 side 0 [7] ; SCL = 0, SDA = 0
134     set pindirs, 1 side 0 [7] ; SCL = 0, SDA = 1
135     set pindirs, 0 side 1 [7] ; SCL = 1, SDA = 0
136     set pindirs, 1 side 1 [7] ; SCL = 1, SDA = 1

```

示例代码对状态机的FIFO执行阻塞式软件IO，以避免配置系统DMA的额外复杂性。例如，I2C启动条件的入队操作如下：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第69至73行

```

69 void pio_i2c_start(PIO pio, uint sm) {
70     pio_i2c_put_or_err(pio, sm, 1u << PIO_I2C_ICOUNT_LSB); // 2条指令的转义代码
71     // 序列
72     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC1_SD0]); // 我们已处于空闲状态
73     // , 仅需将SDA线拉低
74     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC0_SD0]); // 同时拉低时钟线,
75     // 以便传输数据
76 }

```

由于I2C在多个环节可能出错，需能够检测状态机断言的错误标志，清除停止状态并重新启动，在断言停止条件及释放总线之前完成处理。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第15至17行

```
15 bool pio_i2c_check_error(PIO pio, uint sm) {
16     return pio_interrupt_get(pio, sm);
17 }
```

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第19至23行

```
19 void pio_i2c_resume_after_error(PIO pio, uint sm) {
20     pio_sm_drain_tx_fifo(pio, sm);
21     pio_sm_exec(pio, sm, (pio->sm[sm].execctrl & PIO_SM0_EXECCTRL_WRAP_BOTTOM_BITS) >>
22     PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
23     pio_interrupt_clear(pio, sm);
24 }
```

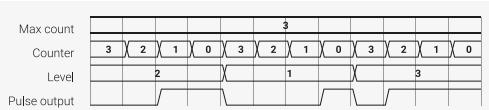
我们需要一些更高级的函数，以便通过FIFO传递格式正确的数据，并在正确位置插入起始位、停止位、NAK等信号。这足以提供与RP2040上其他硬件I2C类似的接口。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c_bus_scan.c 第13至42行

```
13 int main() {
14     stdio_init_all();
15
16     PIO pio = pio0;
17     uint sm = 0;
18     uint offset = pio_add_program(pio, &i2c_program);
19     i2c_program_init(pio, sm, offset, PIN_SDA, PIN_SCL);
20
21     printf("\nPIO I2C总线扫描\n");
22     printf("  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
23
24     for (int addr = 0; addr < (1 << 7); ++addr) {
25         if (addr % 16 == 0) {
26             printf("%02x ", addr);
27         }
28         // 从探针地址执行0字节读取。读取函数
29         // 除最后一个数据字节外，任何时刻返回负结果即表示NAK。
30         // byte. 跳过保留地址。
31         int result;
32         if (reserved_addr(addr))
33             result = -1;
34         else
35             result = pio_i2c_read_blocking(pio, sm, addr, NULL, 0);
36
37         printf(result < 0 ? "." : "@");
38         printf(addr % 16 == 15 ? "\n" : " ");
39     }
40     printf("完成.\n");
41     return 0;
42 }
```

3.6.8. PWM

图56。脉宽调制(PWM)。状态机以固定间隔输出正电压脉冲。这些脉冲的宽度受控，使线路在一定时间内维持高电平(占空比)。其用途之一是通过脉冲频率快于人眼视觉暂留，从而平滑调节LED亮度。



该程序利用Y寄存器反复倒计时至0，并将Y计数与保存在X寄存器中的脉冲宽度进行比较。在计数开始前，输出被置为低电平；当Y的值达到X时，输出被置为高电平。一旦Y归零，该过程重复，输出再次置为低电平。因此，输出处于高电平的时间比例与存储在X中的脉冲宽度成正比。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> 第10至22行

```

10 .program pwm
11 .side_set 1 opt
12
13     pull noblock    side 0 ; 若 FIFO 中有数据，则从 FIFO 拉取至 OSR；否则将 X 复制至 OSR。
14     mov x, osr          ; 将最近拉取的值复制回暂存寄存器 X
15     mov y, isr          ; ISR 包含 PWM 周期，Y 用作计数器。
16 countloop:
17     jmp x!=y noset      ; 如果 X == Y，则将引脚设为高电平，保持两条路径长度匹配
18     jmp skip           side 1
19 noset:
20     nop                  ; 单个占位周期，用以保持两条路径长度一致
21 skip:
22     jmp y-- countloop    ; 循环直到 Y 达到 0，然后从 FIFO 拉取新的 PWM 值

```

通常，PWM 可在特定脉冲宽度下持续数千个脉冲，而无需每次供应新的脉冲宽度。本示例说明了非阻塞 **PULL** (第3.4.7节) 如何实现此功能：若 TX FIFO 为空，非阻塞 **PULL** 会将 X 复制至 OSR。拉取完成后，程序将 OSR 复制回 X，以便与 Y 中的计数值比较。其最终效果是：若本周期开始时 TX FIFO 未提供新的占空比值，则上一周期的占空比 (由失败的 **PULL** 从 X 复制至 OSR，再通过 **MOV** 复制回 X) 将被重复使用，持续所需周期数。

此处展示的另一项实用技术是将ISR用作配置寄存器，前提是无需使用 **IN** 指令。

系统软件可将任意32位数值加载至ISR (通过直接在状态机上执行指令)，并且程序将在每次开始计数时将该数值复制到 Y 寄存器。ISR可用于配置PWM计数范围，状态机的时钟分频器控制计数速率。

要开始调制脉冲，首先需将状态机的侧边引脚映射至目标输出PWM的GPIO，并告知状态机程序在PIO指令存储器中的加载地址：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> 第25至31行

```

25 static inline void pwm_program_init(PIO pio, uint sm, uint offset, uint pin) {
26     pio_gpio_init(pio, pin);
27     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
28     pio_sm_config c = pwm_program_get_default_config(offset);
29     sm_config_set_sideset_pins(&c, pin);
30     pio_sm_init(pio, sm, offset, &c);
31 }

```

加载带有所需计数范围的ISR需要一些准备工作：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第14至20行

```

14 void pio_pwm_set_period(PIO pio, uint sm, uint32_t period) {
15     pio_sm_set_enabled(pio, sm, false);
16     pio_sm_put_blocking(pio, sm, period);
17     pio_sm_exec(pio, sm, pio_encode_pull(false, false));

```

```

18     pio_sm_exec(pio, sm, pio_encode_out(pio_isr, 32));
19     pio_sm_set_enabled(pio, sm, true);
20 }
```

完成上述操作后，状态机即可启用，并可直接向其TX FIFO写入PWM值。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第23至25行

```

23 void pio_pwm_set_level(PIO pio, uint sm, uint32_t level) {
24     pio_sm_put_blocking(pio, sm, level);
25 }
```

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第27行至51行

```

27 int main() {
28     stdio_init_all();
29 #ifndef PICO_DEFAULT_LED_PIN
30 #warning pio/pwm示例需配备带有普通LED的开发板
31     puts("默认LED引脚未定义");
32 #else
33
34 // 待办：获取可用的sm
35 PIO pio = pio0;
36 int sm = 0;
37 uint offset = pio_add_program(pio, &pwm_program);
38 printf("已加载程序，偏移量: %d\n", offset);
39
40 pwm_program_init(pio, sm, offset, PICO_DEFAULT_LED_PIN);
41 pio_pwm_set_period(pio, sm, (1u << 16) - 1);
42
43 int level = 0;
44 while (true) {
45     printf("Level = %d\n", level);
46     pio_pwm_set_level(pio, sm, level * level);
47     level = (level + 1) % 256;
48     sleep_ms(10);
49 }
50#endif
51 }
```

如果 TX FIFO 始终保持填充最新的脉宽值，则该程序将在每个脉冲周期消耗一个新的脉宽值。一旦 FIFO 为空，程序将重新使用最近提供的值。

3.6.9. 加法

尽管 PIO 并非专为计算设计，但只要存在足够长的带子，其极有可能具备图灵完备性。有人推测，在足够高的时钟频率下，它甚至可能运行 DOOM。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.pio> 第7至25行

```

7 .program addition
8
9 ; 从 TX FIFO 弹出两个 32 位整数，将其相加，并将结果推入 TX FIFO。由于正在使用自动推送/拉取
功能，应将其禁用。
11 ; 显示的推拉指令。
12 ;
```

```

13 ; 该程序使用二补码恒等式 x + y == ~(~x - y)
14
15 拉出
16 mov x, ~osr
17 拉出
18 mov y, osr
19 jmp test      ; 该循环等价于以下C语言代码:
20 incr:          ; while (y--)
21 jmp x-- test   ;     x--;
22 test:          ; 此操作最终实现从 x 中减去 y 的效果。
23 jmp y-- incr
24 mov isr, ~x
25 推入

```

在125MHz频率下，完整的32位加法仅需约一分钟。程序从TX FIFO拉取两个数字，并将其和推入RX FIFO，非常适合与系统DMA或处理器直接配合使用：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.c>

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "hardware/pio.h"
12 #include "addition.pio.h"
13
14 // 快答测验：调用此函数时处理器执行了多少次加法？
15 uint32_t do_addition(PIO pio, uint sm, uint32_t a, uint32_t b) {
16     pio_sm_put_blocking(pio, sm, a);
17     pio_sm_put_blocking(pio, sm, b);
18     return pio_sm_get_blocking(pio, sm);
19 }
20
21 int main() {
22     stdio_init_all();
23
24     PIO pio = pio0;
25     uint sm = 0;
26     uint offset = pio_add_program(pio, &addition_program);
27     addition_program_init(pio, sm, offset);
28
29     printf("进行一些随机加法运算: \n");
30     for (int i = 0; i < 10; ++i) {
31         uint a = rand() % 100;
32         uint b = rand() % 100;
33         printf("%u + %u = %u\n", a, b, do_addition(pio, sm, a, b));
34     }
35 }

```

3.6.10. 更多示例

树莓派 Pico 系列 **C/C++ SDK** 手册中设有一章详尽介绍 **PIO**，涵盖本处未呈现的部分软件核心内容。其中包含一个 **PIO + DMA** 逻辑分析仪示例，可在每个周期采样所有 **GPIO**（

带宽近 4Gbps 于 125MHz，尽管这会迅速占用 RP2040 的 RAM)。

Pico 示例仓库的 [pio/](#) 目录中亦有更多示例。

部分较为实验性的示例代码，如 DPI 和 SD 卡支持，现存于 Pico Extras 与 Pico Playground 仓库中。这些 PIO 部分具有功能性，但其周边软件堆栈仍处于实验阶段。

3.7. 寄存器列表

PIO0 和 PIO1 寄存器的地址分别为 `0x50200000` 和 `0x50300000` (在 SDK 中定义为 `PIO0_BASE` 和 `PIO1_BASE`)。

表 367: PIO 寄存器列表

偏移量	名称	说明
0x000	CTRL	PIO 控制寄存器
0x004	FSTAT	FIFO 状态寄存器
0x008	FDEBUG	FIFO 调试寄存器
0x00c	FLEVEL	FIFO 水位
0x010	TXF0	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x014	TXF1	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x018	TXF2	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x01c	TXF3	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x020	RXF0	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x024	RXF1	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x028	RXF2	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。

偏移量	名称	说明
0x02c	RXF3	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x030	IRQ	状态机中断请求标志寄存器。写入 1 可清除。共有 8 个状态机中断请求标志，状态机可对其进行设置、清除和等待。标志与状态机之间无固定对应关系——任意状态机均可使用任何标志。 这 8 个标志中的任意一个均可用于状态机间的时间同步，配合中断（IRQ）和等待（WAIT）指令使用。其中较低的四个标志还被路由至系统级中断请求，与 FIFO 状态中断一并触发——详见如 IRQ0_INTE。
0x034	IRQ_FORCE	向这些位写入 1 会强制触发对应的中断请求（IRQ）。请注意，此操作与 INTF 寄存器不同：此处写入会影响 PIO 的内部状态。INTF 仅断言面向处理器的 IRQ 信号以测试 ISR，状态机不可见该信号。
0x038	INPUT_SYNC_BYPASS	每个 GPIO 输入端均设有 2 级触发器同步器，以防止 PIO 逻辑发生亚稳态。这会增加输入延迟，对于高速同步 IO（如 SPI），可能需要绕过该同步器。该寄存器中的每个位对应一个 GPIO。 0 → 输入已同步（默认） 1 → 同步器已被绕过 如有疑问，请保持此寄存器全零。
0x03c	DBG_PADOUT	读取以采样 PIO 当前驱动至 GPIO 的引脚输出值。RP2040 共有 30 个 GPIO，最高两位硬连线为 0。
0x040	DBG_PADOE	读取以采样 PIO 当前驱动至 GPIO 的引脚输出使能（方向）。RP2040 共有 30 个 GPIO，最高两位硬连线为 0。
0x044	DBG_CFGINFO	PIO 硬件具有一些在不同芯片产品之间可能变化的自由参数。 这些参数应在芯片数据手册中提供，但此处亦有展示。
0x048	INSTR_MEM0	指令存储位置 0 的写入专用访问
0x04c	INSTR_MEM1	指令存储位置 1 的写入专用访问
0x050	INSTR_MEM2	指令存储位置 2 的写入专用访问
0x054	INSTR_MEM3	指令存储位置 3 的写入专用访问
0x058	INSTR_MEM4	指令存储位置 4 的写入专用访问
0x05c	INSTR_MEM5	指令存储位置 5 的写入专用访问
0x060	INSTR_MEM6	指令存储位置 6 的写入专用访问
0x064	INSTR_MEM7	指令存储位置 7 的写入专用访问

偏移量	名称	说明
0x068	INSTR_MEM8	指令存储位置 8 的写入专用访问
0x06c	INSTR_MEM9	指令存储位置 9 的写入专用访问
0x070	INSTR_MEM10	指令存储位置 10 的写入专用访问
0x074	INSTR_MEM11	指令存储位置 11 的写入专用访问
0x078	INSTR_MEM12	指令存储位置 12 的写入专用访问
0x07c	INSTR_MEM13	指令存储位置 13 的写入专用访问
0x080	INSTR_MEM14	指令存储位置 14 的写入专用访问
0x084	INSTR_MEM15	指令存储位置 15 的写入专用访问
0x088	INSTR_MEM16	对指令内存地址16的仅写访问
0x08c	INSTR_MEM17	对指令内存地址17的仅写访问
0x090	INSTR_MEM18	对指令内存地址18的仅写访问
0x094	INSTR_MEM19	对指令内存地址19的仅写访问
0x098	INSTR_MEM20	对指令内存地址20的仅写访问
0x09c	INSTR_MEM21	对指令内存地址21的仅写访问
0x0a0	INSTR_MEM22	对指令内存地址22的仅写访问
0x0a4	INSTR_MEM23	对指令内存地址23的仅写访问
0x0a8	INSTR_MEM24	对指令内存地址24的仅写访问
0x0ac	INSTR_MEM25	对指令内存地址25的仅写访问
0x0b0	INSTR_MEM26	对指令内存地址26的仅写访问
0x0b4	INSTR_MEM27	对指令内存地址27的仅写访问
0x0b8	INSTR_MEM28	对指令内存地址28的仅写访问
0x0bc	INSTR_MEM29	对指令内存地址29的仅写访问
0x0c0	INSTR_MEM30	对指令内存地址30的仅写访问
0x0c4	INSTR_MEM31	对指令内存地址31的仅写访问
0x0c8	SM0_CLKDIV	状态机0的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0cc	SM0_EXECCTRL	状态机0的执行/行为设置
0x0d0	SM0_SHIFTCTRL	控制状态机0输入/输出移位寄存器的行为
0x0d4	SM0_ADDR	状态机0的当前指令地址
0x0d8	SM0_INSTR	读取以查看状态机当前指向的指令 0号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x0dc	SM0_PINCTRL	状态机引脚控制
0x0e0	SM1_CLKDIV	状态机1的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0e4	SM1_EXECCTRL	状态机1的执行/行为设置

偏移量	名称	说明
0x0e8	SM1_SHIFTCTRL	控制状态机1输入/输出移位寄存器的行为
0x0ec	SM1_ADDR	状态机1的当前指令地址
0x0f0	SM1_INSTR	读取以查看状态机当前指向的指令 1号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x0f4	SM1_PINCTRL	状态机引脚控制
0x0f8	SM2_CLKDIV	状态机2的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0fc	SM2_EXECCTRL	状态机2的执行/行为设置
0x100	SM2_SHIFTCTRL	控制状态机2输入/输出移位寄存器的行为
0x104	SM2_ADDR	状态机2的当前指令地址
0x108	SM2_INSTR	读取以查看状态机当前指向的指令 2号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x10c	SM2_PINCTRL	状态机引脚控制
0x110	SM3_CLKDIV	状态机3的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x114	SM3_EXECCTRL	状态机3的执行/行为设置
0x118	SM3_SHIFTCTRL	控制状态机3输入/输出移位寄存器的行为
0x11c	SM3_ADDR	状态机3的当前指令地址
0x120	SM3_INSTR	读取状态机3程序计数器当前指向的指令 写入以立即执行指令（包括跳转）并随后恢复执行。
0x124	SM3_PINCTRL	状态机引脚控制
0x128	INTR	原始中断
0x12c	IRQ0_INTE	irq0中断使能
0x130	IRQ0_INTF	irq0中断强制
0x134	IRQ0_INTS	irq0中断屏蔽及强制后的状态
0x138	IRQ1_INTE	irq1中断使能
0x13c	IRQ1_INTF	irq1中断强制
0x140	IRQ1_INTS	irq1中断屏蔽及强制后的状态

PIO: CTRL寄存器

偏移: 0x000

描述

PIO控制寄存器

表368. CTRL
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:8	<p>CLKDIV_RESTART: 重新启动状态机的时钟分频器，起始相位为0。时钟分频器为自由运行模式，启动后，其输出（包括分数抖动）完全由SMx_CLKDIV中配置的整数/分数分频系数决定。这意味着，若同时通过向此字段写入多个“1”位重新启动多个具有相同分频系数的时钟分频器，则这些状态机的执行时钟将实现精确同步。</p> <p>请注意，设置或清除SM_ENABLE并不会停止时钟分频器的运行，因此一旦多个状态机的时钟同步，禁用或重新启用状态机是安全的，同时保持时钟分频器同步。</p> <p>另请注意，可在状态机运行时写入CLKDIV_RESTART，此操作有助于在动态更改除数(SMx_CLKDIV)后重新同步时钟分频器。</p>	SC	0x0
7:4	<p>SM_RESTART: 写入1可立即清除内部状态机状态，该状态通常难以访问且会影响后续执行。</p> <p>具体而言，以下内容将被清除：输入和输出移位计数器；输入移位寄存器内容；延迟计数器；等待IRQ状态；任何写入至SMx_INSTR或由OUT/MOV EXEC执行的停滞指令；因OUT_STICKY导致仍保持断言状态的引脚写入。</p> <p>程序计数器、输出移位寄存器内容及X/Y临时寄存器不受影响。</p>	SC	0x0
3:0	<p>SM_ENABLE: 通过向这四个位分别写入1或0，启用或禁用各状态机。禁用状态机后，除系统直接写入SMx_INSTR的指令外，该状态机将停止执行指令。</p> <p>可同时设置或清除多个位，以同步启动或停止多个状态机。</p>	读写	0x0

PIO: FSTAT寄存器

偏移量: 0x004

描述

FIFO状态寄存器

表369: FSTAT
寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	TXEMPTY : 状态机的TX FIFO为空	只读	0xf
23:20	保留。	-	-
19:16	TXFULL : 状态机的TX FIFO已满	只读	0x0
15:12	保留。	-	-
11:8	RXEMPTY : 状态机的RX FIFO为空	只读	0xf
7:4	保留。	-	-

位	描述	类型	复位值
3:0	RXFULL : 状态机的RX FIFO已满	只读	0x0

PIO: FDEBUG 寄存器

偏移量: 0x008

描述

FIFO调试寄存器

表 370: FDEBUG 寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	TXSTALL : 状态机在阻塞型PULL操作或启用自动拉取的OUT操作时, 因TX FIFO为空而暂停。写入1以清除该状态。	WC	0x0
23:20	保留。	-	-
19:16	TXOVER : 发生了TX FIFO溢出(即系统在FIFO满时尝试写入)。 写入1以清除该标志。请注意, 写入满时操作不会以任何方式改变FIFO的状态或内容, 但系统尝试写入的数据将被丢弃, 因此若该标志被置位, 您的软件很可能已丢失部分数据。	WC	0x0
15:12	保留。	-	-
11:8	RXUNDER : 发生了RX FIFO欠载(即系统在FIFO为空时尝试读取)。 写入1以清除该标志。请注意, 读取空时操作不会影响FIFO的状态, 但从空FIFO读取的数据是未定义的, 因此该标志通常仅因软件故障而置位。	WC	0x0
7:4	保留。	-	-
3:0	RXSTALL : 状态机在阻塞PUSH操作过程中, 或启用自动推送的IN操作时, 因RX FIFO已满而暂停。当对满FIFO执行非阻塞PUSH操作时, 该标志亦会置位, 此时状态机已丢弃数据。写入1以清除该标志。	WC	0x0

PIO: FLEVEL 寄存器

偏移: 0x00c

描述

FIFO水位

表 371: FLEVEL 寄存器

位	描述	类型	复位值
31:28	RX3	只读	0x0
27:24	TX3	只读	0x0
23:20	RX2	只读	0x0
19:16	TX2	只读	0x0
15:12	RX1	只读	0x0
11:8	TX1	只读	0x0
7:4	RX0	只读	0x0
3:0	TX0	只读	0x0

PIO: TXF0、TXF1、TXF2、TXF3寄存器

偏移: 0x010、0x014、0x018、0x01c

表372. TXF0、TXF1、TXF2、TXF3寄存器

位	描述	类型	复位值
31:0	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。	WF	0x00000000

PIO: RXF0、RXF1、RXF2、RXF3寄存器

偏移: 0x020、0x024、0x028、0x02c

表373. RXF0、RXF1、RXF2、RXF3寄存器

位	描述	类型	复位值
31:0	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空的 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBU G_RXUNDER 错误标志。 从空的 FIFO 读取返回给系统的数据为未定义。	RF	-

PIO: IRQ 寄存器

偏移: 0x030

表 374. IRQ 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	状态机中断请求标志寄存器。写入 1 可清除。共有 8 个状态机 IRQ 标志，可由状态机设置、清除及等待。 标志与状态机之间无固定对应关系——任意状态机均可使用任何标志。 这 8 个标志中的任意一个均可用于状态机间的时间同步，配合中断（IRQ）和等待（WAIT）指令使用。其中较低的四个标志还被路由至系统级中断请求，与 FIFO 状态中断一并触发——详见如 IRQ0_INTE。	WC	0x00

PIO: IRQ_FORCE 寄存器

偏移: 0x034

表 375. IRQ_FORCE 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	向这些位写入 1 将强制断言相应的 IRQ。 注意，此寄存器与 INTF 寄存器不同：在此写入会影响 PIO 内部状态。INTF 仅断言面向处理器的 IRQ 信号以测试 ISR，状态机不可见该信号。	WF	0x00

PIO: INPUT_SYNC_BYPASS 寄存器

偏移量: 0x038

表 376.
INPUT_SYNC_BYPASS
寄存器

位	描述	类型	复位值
31:0	<p>每个GPIO输入端均设有2级触发器同步器，以防止PIO逻辑发生亚稳态。这会增加输入延迟，对于高速同步IO（如SPI），可能需要绕过该同步器。该寄存器中的每个位对应一个GPIO。</p> <p>0 → 输入已同步（默认） 1 → 同步器已被绕过 如有疑问，请保持此寄存器全零。</p>	读写	0x00000000

PIO: DBG_PADOUT 寄存器

偏移量: 0x03c

表 377。
DBG_PADOUT 寄存器

位	描述	类型	复位值
31:0	读取以采样PIO当前驱动至GPIO的引脚输出值。RP2040共有30个GPIO，最高两位硬连线为0。	只读	0x00000000

PIO: DBG_PADOE 寄存器

偏移量: 0x040

表 378。
DBG_PADOE 寄存器

位	描述	类型	复位值
31:0	读取以采样PIO当前驱动至GPIO的引脚输出使能（方向）。RP2040共有30个GPIO，最高两位硬连线为0。	只读	0x00000000

PIO: DBG_CFGINFO 寄存器

偏移量: 0x044

描述

PIO 硬件包含某些自由参数，可能因芯片型号而异。

这些参数应在芯片数据手册中提供，但此处亦有展示。

表 379。
DBG_CFGINFO
寄存器

位	描述	类型	复位值
31:22	保留。	-	-
21:16	IMEM_SIZE : 指令存储器大小，单位为单条指令	只读	-
15:12	保留。	-	-
11:8	SM_COUNT : 该 PIO 实例所配置的状态机数量。	只读	-
7:6	保留。	-	-
5:0	FIFO_DEPTH : 状态机 TX/RX FIFO 的深度，单位为字。 通过 SHIFTCTRL_FJOIN 连接 FIFO，可形成深度加倍的单一 FIFO。	只读	-

PIO: INSTR_MEM0, INSTR_MEM1, ..., INSTR_MEM30, INSTR_MEM31 寄存器

偏移量: 0x048, 0x04C, ..., 0x0C0, 0x0C4

表 380。
`INSTR_MEM0`,
`INSTR_MEM1`, ...,
`INSTR_MEM30`,
`INSTR_MEM31`
 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	对指令存储位置的只写访问 N	WO	0x0000

PIO: SM0_CLKDIV、SM1_CLKDIV、SM2_CLKDIV、SM3_CLKDIV 寄存器

偏移量: 0x0c8、0x0e0、0x0f8、0x110

描述

状态机的时钟分频寄存器 N

频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)

表 381。
`SM0_CLKDIV`,
`SM1_CLKDIV`,
`SM2_CLKDIV`,
`SM3_CLKDIV`
 寄存器

位	描述	类型	复位值
31:16	INT : 有效频率为 $\text{sysclk}/(\text{int} + \text{frac}/256)$ 。 值为0时解释为65536。若 INT 为 0，则 FRAC 亦须为 0。	读写	0x0001
15:8	FRAC : 时钟分频的分数部分	读写	0x00
7:0	保留。	-	-

PIO: SM0_EXECCTRL、SM1_EXECCTRL、SM2_EXECCTRL、SM3_EXECCTRL 寄存器

偏移量: 0x0cc、0x0e4、0x0fc、0x114

描述

状态机的执行/行为设置 N

表 382。
`SM0_EXECCTRL`,
`SM1_EXECCTRL`,
`SM2_EXECCTRL`,
`SM3_EXECCTRL`
 寄存器

位	描述	类型	复位值
31	EXEC_STALLED : 若为1，则写入 <code>SMx_INSTR</code> 的指令被阻塞并由状态机锁存。该指令完成后将被清零。	只读	0x0
30	SIDE_EN : 若为1，Delay/Side-set 指令字段的最高有效位用作侧设使能位，而非侧设数据位。此设置允许指令选择性地执行侧设，而非每条指令均执行，但最大侧设宽度由5位减至4位。请注意， <code>PINCTRL_SIDESET_COUNT</code> 的值包含此使能位。	读写	0x0
29	SIDE_PINDIR : 若为1，侧设数据应用于引脚方向，而非引脚电平。	读写	0x0
28:24	JMP_PIN : 用于 <code>JMP PIN</code> 条件的 GPIO 编号，不受输入映射影响。	读写	0x00
23:19	OUT_EN_SEL : 选择用于内联OUT使能的数据位	读写	0x00
18	INLINE_OUT_EN : 若为1，则使用OUT数据中的某个位作为辅助写使能与OUT_STICKY同时使用时，使能为0的写操作将取消最近一次的引脚写入。这可以实现有效的掩码或覆盖行为 基于状态机引脚写入的优先级顺序 ($SM0 < SM1 < ...$)	读写	0x0
17	OUT_STICKY : 持续对引脚断言最近的OUT/SET信号	读写	0x0
16:12	WRAP_TOP : 达到此地址后，程序执行跳转至wrap_bottom。 若指令为跳转且跳转条件成立，则跳转具有优先权。	读写	0x1f

位	描述	类型	复位值
11:7	WRAP_BOTTOM : 达到wrap_top后，程序执行跳转至此地址。	读写	0x00
6:5	保留。	-	-
4	STATUS_SEL : 用于MOV x, STATUS指令中的比较选择。	读写	0x0
	枚举值:		
	0x0 → TXLEVEL: 当TX FIFO级别 < N时，返回全1，否则返回全0		
	0x1 → RXLEVEL: 当RX FIFO级别 < N时，返回全1，否则返回全0		
3:0	STATUS_N : MOV x, STATUS 指令的比较等级	读写	0x0

PIO: SM0_SHIFTCTRL, SM1_SHIFTCTRL, SM2_SHIFTCTRL, SM3_SHIFTCTRL 寄存器

偏移量: 0x0d0, 0x0e8, 0x100, 0x118

描述

控制状态机 N 输入/输出移位寄存器的行为

表383。
SM0_SHIFTCTRL,
SM1_SHIFTCTRL,
SM2_SHIFTCTRL,
SM3_SHIFTCTRL
寄存器

位	描述	类型	复位值
31	FJOIN_RX : 当该位为1时，接收FIFO占用发送FIFO的存储空间，深度加倍。 因此，发送FIFO被禁用（始终读作既满又空）。 该位改变时，FIFO将被清空。	读写	0x0
30	FJOIN_TX : 当该位为1时，发送FIFO占用接收FIFO的存储空间，深度加倍。 因此，接收FIFO被禁用（始终读作既满又空）。 该位改变时，FIFO将被清空。	读写	0x0
29:25	PULL_THRESH : 在自动拉取或条件拉取（空时拉取）发生前，从OS R中移出的位数。 写入值0表示32。	读写	0x00
24:20	PUSH_THRESH : 在执行自动推送（autopush）或条件推送（PUSH IFFULL）前，移入ISR的比特数。 写入值0表示32。	读写	0x00
19	OUT_SHIFTDIR : 1 = 从输出移位寄存器向右移位；0 = 向左移位。	读写	0x1
18	IN_SHIFTDIR : 1 = 输入移位寄存器向右移位（数据从左侧进入）；0 = 向左移位。	读写	0x1
17	AUTOPULL : 当输出移位寄存器清空时自动拉取，即在执行OUT指令且输出移位计数器达到或超过PULL_THRESH时。	读写	0x0
16	AUTOPUSH : 当输入移位寄存器填满时自动推送，即在执行IN指令导致输入移位计数器达到或超过PUSH_THRESH时。	读写	0x0
15:0	保留。	-	-

PIO: SM0_ADDR、SM1_ADDR、SM2_ADDR、SM3_ADDR寄存器

偏移: 0x0d4、0x0ec、0x104、0x11c

表384。SM0_ADDR,
SM1_ADDR,
SM2_ADDR,
SM3_ADDR寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	状态机当前指令地址 N	只读	0x00

PIO: SM0_INSTR, SM1_INSTR, SM2_INSTR, SM3_INSTR寄存器

偏移量: 0x0d8, 0x0f0, 0x108, 0x120

表385。
SM0_INSTR,
SM1_INSTR,
SM2_INSTR,
SM3_INSTR寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	读取表示状态机 N程序计数器当前指向的指令。 写入以立即执行指令（包括跳转）并随后恢复执行。	读写	-

PIO: SM0_PINCTRL, SM1_PINCTRL, SM2_PINCTRL, SM3_PINCTRL寄存器

偏移量: 0x0dc, 0x0f4, 0x10c, 0x124

描述

状态机引脚控制

表386。
SM0_PINCTRL,
SM1_PINCTRL,
SM2_PINCTRL,
SM3_PINCTRL寄存器

位	描述	类型	复位值
31:29	SIDESET_COUNT: 用于side-set的延迟/side-set指令字段最高有效位数。包括启用位（如存在）。最小为0（所有为延迟位，无侧置），最大为5（所有为侧置，无延迟）。	读写	0x0
28:26	SET_COUNT: 由SET断言的引脚数量，范围为0至5（含）。	读写	0x5
25:20	OUT_COUNT: 由OUT PINS、OUT PINDIRS或MOV PINS指令断言的引脚数量。范围为0至32（含）。	读写	0x00
19:15	IN_BASE: 映射至状态机IN数据总线最低有效位的引脚编号。编号较高的引脚依次映射至更高有效位，且对引脚编号取模32。	读写	0x00
14:10	SIDESET_BASE: 受侧置操作影响的最低编号引脚。指令侧置/延迟字段的最高有效位（最多5位，由SIDESET_COUNT确定）用于侧置数据，其余最低有效位用于延迟计数。side-set部分的最低有效位写入该引脚，更高有效位写入编号更高的引脚。	读写	0x00
9:5	SET_BASE: 将被SET PINS或SET PINDIRS指令影响的编号最低的引脚。写入该引脚的数据为SET数据的最低有效位。	读写	0x00
4:0	OUT_BASE: 将被OUT PINS、OUT PINDIRS或MOV PINS指令影响的编号最低的引脚。写入该引脚的数据始终为OUT或MOV数据的最低有效位。	读写	0x00

PIO: INTR 寄存器

偏移: 0x128

描述

原始中断

表 387. INTR
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

PIO：IRQ0_INTE 寄存器

偏移: 0x12c

描述

irq0中断使能

表 388. IRQ0_INTE
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ0_INTF 寄存器

偏移: 0x130

描述

irq0中断强制

表389. IRQ0_INTF
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO: IRQ0_INTS 寄存器

偏移量: 0x134

说明

irq0中断屏蔽及强制后的状态

表390. IRQ0_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

PIO: IRQ1_INTE 寄存器

偏移: 0x138

描述

irq1中断使能

表391. IRQ1_INTF
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ1_INTF 寄存器

偏移: 0x13c

描述

irq1中断强制

表392. IRQ1_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ1_INTS 寄存器

偏移: 0x140

说明

irq1中断屏蔽及强制后的状态

表393. IRQ1_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

第4章 外设

4.1. USB

4.1.1. 概述

先决知识

本节内容需要具备USB协议相关知识。若您对本节所用术语不熟悉，建议参考《USB Made Simple》一书。

RP2040内置USB 2.0控制器，支持下列任一模式运行：

- 全速设备（12Mbps）
- 能够与低速（1.5Mbps）和全速设备通信的主机。包括连接到USB集线器的多个下游设备。

集成了USB 1.1 PHY，用于将USB控制器与芯片的 DP 和 DM 引脚接口连接。

4.1.1.1. 功能特性

USB控制器硬件负责处理底层USB协议，程序员的主要任务是配置控制器，并在总线事件发生时提供或使用数据缓冲区。当控制器需要处理器响应时，会触发中断。USB控制器配备4kB的DPSRAM，用于配置和数据缓冲。

4.1.1.1.1. 设备模式

- 兼容USB 2.0的全速设备（12Mbps）
- 支持最多32个端点（端点0 → 15，含输入和输出方向）
- 支持控制、等时、批量和中断端点类型
- 支持双缓冲
- DPSRAM中可用缓冲区空间为3840字节，相当于60个64字节的缓冲区。

4.1.1.1.2. 主机模式

- 能够与全速（12Mbps）及低速（1.5Mbps）设备通信
- 能够通过USB集线器与多个设备通信，包括连接至全速集线器的低速设备
- 硬件支持轮询最多15个中断端点。（中断端点用于集线器通知主机连接/断开事件，鼠标通知主机移动等。）

4.1.2. 架构

4.1.2.1. 时钟频率

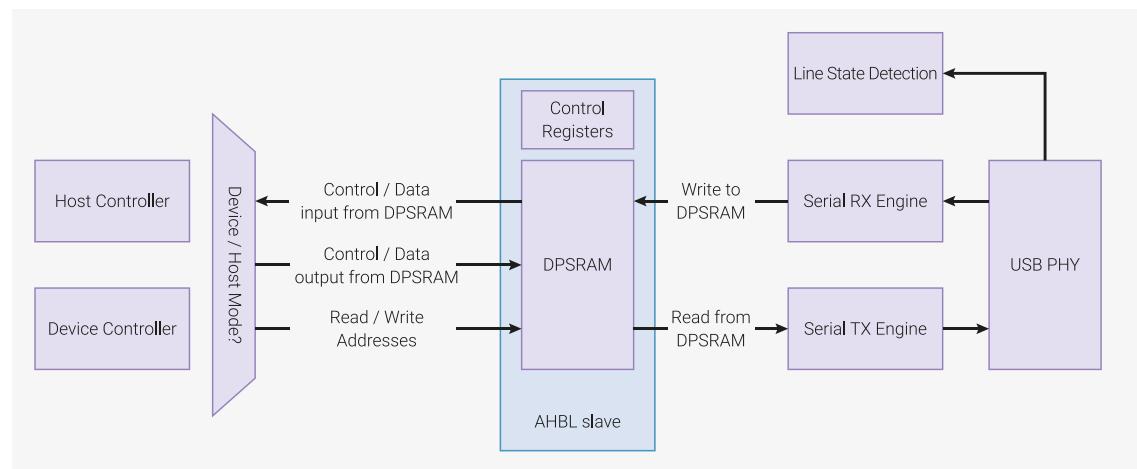
该控制器要求 `clk_usb` 以48MHz运行。

注意

`clk_sys` 也必须运行于高于48MHz的频率。详见RP2040-E16。

4.1.2.2. 概述

图57。USB控制器架构的简化概述
。



USB控制器是一种高效节省空间的设计，将设备控制器或主机控制器复用在一组通用组件上。各组件详述如下。

4.1.2.3. USB PHY

USB PHY 提供 USB `DP`与 `DM`引脚和控制器数字逻辑之间的电气接口。`DP`与 `DM`引脚成差分对，意味着其数值始终相反，除非用于编码特定线路状态（如`SE0`等）。USB PHY 驱动 `DP`与 `DM`引脚以传输数据，同时负责对任何输入数据进行差分接收。USB PHY 向线路状态检测模块提供单端和差分接收的数据。

USB PHY 内置上拉电阻与下拉电阻。若控制器作为全速设备，`DP`引脚会被上拉，以向主机指示已连接全速设备。在主机模式下，`DP`与 `DM`引脚施加弱下拉，使线路保持逻辑低电平，直到设备上拉 `DP`以表示全速，或上拉 `DM`以表示低速。

4.1.2.4. 线路状态检测

[2] 定义了若干需要检测的线路状态（总线复位、已连接、挂起、恢复、数据1、数据0等）。线路状态检测模块配备多个状态机，用于检测这些状态并向其他硬件组件发送事件信号。USB 中不存在共享时钟信号，因此接收数据必须由内部时钟采样。USB 全速的最大数据传输速率为 12Mbps。接收数据以 48MHz 频率采样，提供 4 个时钟周期用于捕获和滤波总线状态。线路状态检测模块将滤波后的接收数据分发至串行接收引擎。

4.1.2.5. 串行接收引擎

串行接收引擎对线路状态检测模块捕获的接收数据进行解码。其输出以下信息：

- 传入数据包的 **PID**
- 传入数据的设备地址
- 传入数据的设备端点
- 数据字节

串行接收引擎通过对传入数据执行CRC校验，检测RX数据中的错误。任何错误都会向其他硬件模块发出信号，并可能触发中断。

i 注意

如果在主机模式或设备模式的数据包传输过程中断开USB线缆，硬件会报告错误。若启用错误中断，您的软件必须考虑该场景。

4.1.2.6 串行发送引擎

串行发送引擎是串行接收引擎的镜像。它连接到当前活动的控制器（设备端或主机端）。它生成 **TOKEN** 和 **DATA** 数据包，包含CRC计算，并在总线上发送这些数据包。

4.1.2.7 DPSRAM

USB控制器配备4KB（4096字节）DPSRAM（双端口静态随机存取存储器）。DPSRAM用于存储控制寄存器和数据缓冲区。DPSRAM可作为32位宽内存，从USB控制器地址0（**0x5010000**）访问。

DPSRAM具有以下特性，这些特性与RP2040上的大多数寄存器不同：

- 支持8位、16位和32位访问。寄存器通常仅支持32位访问。
- DPSRAM不支持设置/清除别名功能。RP2040寄存器通常支持这些功能。

数据缓冲区通常为64字节，这是大多数FS数据包的最大常规包大小。对于等时端点，支持最大1023字节的缓冲区大小。对于其他数据包类型，每个缓冲区的最大大小为64字节。

4.1.2.7.1. 并发访问

USB控制器中的DPSRAM应视为异步且非原子操作。它是一种双端口SRAM，这意味着处理器拥有一个端口用于读写内存，USB控制器也拥有一个端口用于读写内存。这意味着处理器和USB控制器可以同时访问相同的内存地址。一方可能正在写入，另一方可能正在读取。如果控制器在处理器写入内存时读取内存，可能会导致数据不一致。必须谨慎以避免此类情况发生。

缓冲区控制寄存器中的 **AVAILABLE** 位用于指示缓冲区的所有权归属。处理器应将该位设置为1，以将缓冲区所有权交予控制器。控制器在使用缓冲区后，会将该位重新置0。应将 **AVAILABLE** 位与缓冲区控制寄存器中的其他数据分开设置，以确保在设置 **AVAILABLE** 位时，缓冲区控制寄存器中的其余数据保持准确。

这是因为处理器时钟 **clk_sys** 可能远快于 **clk_usb** 时钟。

因此，**clk_sys** 可能在USB控制器以较慢时钟读取数据时更新数据。正确的操作流程是：

- 将缓冲区信息（长度等）写入缓冲区控制寄存器

- 执行若干 `nop` 指令，经过 `clk_sys` 周期，以确保至少经过一个 `clk_usb` 周期。例如，若 `clk_sys` 运行于 125MHz，`clk_usb` 运行于 48MHz，则 125/48 向上取整即为 3 条 `nop` 指令。

- **设置 AVAILABLE 位**

若 `clk_sys` 与 `clk_usb` 频率相同，则无需单独设置 `AVAILABLE` 位。

i 注意

当控制器向 DPSRAM 回写状态时，会对缓冲区 0 的低两字节和缓冲区 1 的高两字节执行 16 位写操作。因此，若使用双缓冲模式，软件更新缓冲区控制寄存器时，将其视为两个 16 位寄存器处理最为安全。

4.1.2.7.2. 布局

地址 `0x0` 至 `0xff` 区域用于存储包含配置信息的控制寄存器。剩余地址空间 `0x100` 至 `0xffff`（共 3840 字节）可用作数据缓冲区。控制器具有从地址 `0x10000` 开始的控制寄存器。

内存布局根据控制器是处于设备模式还是主机模式而不同。在设备模式下，主机可以访问多个端点，因此每个端点必须配置端点控制寄存器和缓冲区控制寄存器。

在主机模式下，运行于处理器上的主机软件负责决定访问哪些端点和设备，因此仅需一组端点控制寄存器和缓冲区控制寄存器。除软件驱动的传输外，主机控制器可轮询最多 15 个中断端点，并为每个中断端点配备相应寄存器。

表394. DPSRAM
布局

偏移量	设备功能	主机功能
0x0	设置包（8字节）	
0x8	EP1输入控制	中断端点控制1
0xc	EP1输出控制	备用
0x10	EP2输入控制	中断端点控制2
0x14	EP2输出控制	备用
0x18	EP3输入控制	中断端点控制3
0x1c	EP3输出控制	备用
0x20	EP4 输入控制	中断端点控制 4
0x24	EP4 输出控制	备用
0x28	EP5 输入控制	中断端点控制 5
0x2c	EP5 输出控制	备用
0x30	EP6 输入控制	中断端点控制 6
0x34	EP6 输出控制	备用
0x38	EP7 输入控制	中断端点控制 7
0x3c	EP7 输出控制	备用
0x40	EP8 输入控制	中断端点控制 8
0x44	EP8 输出控制	备用
0x48	EP9 输入控制	中断端点控制 9
0x4c	EP9 出端控制	备用
0x50	EP10 入端控制	中断端点控制 10
0x54	EP10 出端控制	备用

偏移量	设备功能	主机功能
0x58	EP11 入端控制	中断端点控制 11
0x5c	EP11 出端控制	备用
0x60	EP12 入端控制	中断端点控制 12
0x64	EP12 出端控制	备用
0x68	EP13 入端控制	中断端点控制 13
0x6c	EP13 出端控制	备用
0x70	EP14 入端控制	中断端点控制 14
0x74	EP14 出端控制	备用
0x78	EP15 入端控制	中断端点控制 15
0x7c	EP15 出口控制	备用
0x80	EP0 输入缓冲控制	EPx 缓冲控制
0x84	EP0 出口缓冲控制	备用
0x88	EP1 输入缓冲控制	中断端点缓冲控制 1
0x8c	EP1 出口缓冲控制	备用
0x90	EP2 输入缓冲控制	中断端点缓冲控制 2
0x94	EP2 出口缓冲控制	备用
0x98	EP3 输入缓冲控制	中断端点缓冲控制 3
0x9c	EP3 出口缓冲控制	备用
0xa0	EP4 输入缓冲控制	中断端点缓冲控制 4
0xa4	EP4 出口缓冲控制	备用
0xa8	EP5 输入缓冲控制	中断端点缓冲控制 5
0xac	EP5 出端点缓冲区控制	备用
0xb0	EP6 入端点缓冲区控制	中断端点缓冲区控制 6
0xb4	EP6 出端点缓冲区控制	备用
0xb8	EP7 入端点缓冲区控制	中断端点缓冲区控制 7
0xbc	EP7 出端点缓冲区控制	备用
0xc0	EP8 入端点缓冲区控制	中断端点缓冲区控制 8
0xc4	EP8 出端点缓冲区控制	备用
0xc8	EP9 入端点缓冲区控制	中断端点缓冲区控制 9
0xcc	EP9 出端点缓冲区控制	备用
0xd0	EP10 入端点缓冲区控制	中断端点缓冲区控制 10
0xd4	EP10 输出缓冲区控制	备用
0xd8	EP11 输入缓冲区控制	中断端点缓冲区控制 11
0xdc	EP11 输出缓冲区控制	备用
0xe0	EP12 输入缓冲区控制	中断端点缓冲区控制 12
0xe4	EP12 输出缓冲区控制	备用

偏移量	设备功能	主机功能
0xe8	EP13 输入缓冲区控制	中断端点缓冲区控制 13
0xec	EP13 输出缓冲区控制	备用
0xf0	缓冲区控制中的 EP14	中断端点缓冲区控制 14
0xf4	EP14 出缓冲区控制	备用
0xf8	缓冲区控制中的 EP15	中断端点缓冲区控制 15
0xfc	EP15 出缓冲区控制	备用
0x100	EP0 缓冲区 0 (输入和输出共用)	端点控制
0x140	可选的 EP0 缓冲区 1	备用
0x180	数据缓冲区	

4.1.2.7.3. 端点控制寄存器

端点控制寄存器用于配置端点，内容包括：

- 端点类型
- 其数据缓冲区的基地址，或如为双缓冲则为数据缓冲区
- 端点应触发的中断事件

设备必须支持端点0，以响应SETUP数据包并完成枚举。因此，EP0没有端点控制寄存器。其缓冲区起始地址为 **0x100**。所有其他端点均可配置为单缓冲或双缓冲，并映射到已编程地址。由于EP0无端点控制寄存器，EP0的中断使能控制来自SIE_CTRL。

表395。端点
控制寄存器布局

位域	设备功能	主机功能
31	端点使能	
30	单缓冲 (64字节) = 0, 双缓冲 (64字节 x 2) = 1	
29	每传输一个缓冲区即使能中断	
28	每传输两个缓冲区 (仅适用于双缓冲) 即使能中断	
27:26	端点类型：控制 = 0, 等时传输 = 1, 批量传输 = 2, 中断传输 = 3	
25:18	不适用	主机控制器轮询此端点的周期。仅适用于中断端点，以毫秒为单位减1指定。例如：值9表示每10毫秒轮询一次此端点。
17	因停滞引发的中断	
16	因NAK响应引发的中断	
15:6	DPSRAM中数据缓冲区的地址基准偏移	

注意

数据缓冲区地址必须按64字节对齐，因位0到5被忽略

4.1.2.7.4. 缓冲区控制寄存器

缓冲区控制寄存器包含该端点数据缓冲区状态的信息。该寄存器由处理器与控制器共享。若端点配置为单缓冲，则仅使用缓冲区前半部分（位0至15）。

若为双缓冲，缓冲区选择从缓冲区0开始。此后，缓冲区选择在缓冲区0和1间切换，除非“重置缓冲区选择”位被置位（此操作会将缓冲区选择重置为缓冲区0）。缓冲区选择的值为控制器内部状态，处理器无法访问。

对于主机中断及EPx上的等时包，即便传输未成功完成，缓冲区满位仍会在完成时被置位。可读取 SIE_STATUS 寄存器中的错误位以确定错误。

表 396。缓冲区控制寄存器布局

位域	功能
31	缓冲区 1 已满。处理器应将其设置为 1，表示 IN 传输；设置为 0，表示 OUT 传输。控制器将此位设置为 1，表示 OUT 传输，因为缓冲区已填满。 控制器将此位设置为 0，表示 IN 传输，因为缓冲区已清空。仅适用于双缓冲模式
30	缓冲区 1 的传输最后缓冲区标志——仅适用于双缓冲模式
29	缓冲区 1 的数据 PID——DATA0 = 0, DATA1 = 1——仅适用于双缓冲模式
27:28	同步模式的双缓冲区偏移（0 = 128, 1 = 256, 2 = 512, 3 = 1024）
26	缓冲区 1 可用。指示缓冲区是否可被控制器用于传输。缓冲区配置完成后，处理器将其设置为 1。控制器在使用完缓冲区后将其设置为 0，即已将数据发送至主机进行 IN 传输，或已将主机数据填充进缓冲区进行 OUT 传输。仅适用于双缓冲模式。
25:16	缓冲区1传输长度——仅适用于双缓冲模式
15	缓冲区0已满。处理器应将其设置为 1，表示 IN 传输；设置为 0，表示 OUT 传输。控制器将此位设置为 1，表示 OUT 传输，因为缓冲区已填满。 控制器在 IN 传输时将其设置为 0，因为缓冲区已被清空。
14	缓冲区0的最后一个传输缓冲区
13	缓冲区0的数据PID——DATA0 = 0, DATA1 = 1
12	将缓冲区选择重置为缓冲区0——在传输结束时清除。仅适用于设备。
11	设备发送STALL，主机接收STALL。
10	缓冲区0可用。指示缓冲区是否可被控制器用于传输。缓冲区配置完成后，处理器将其设置为 1。控制器在使用完缓冲区后将其设置为 0，即已将数据发送至主机进行 IN 传输，或已将主机数据填充进缓冲区进行 OUT 传输。
9:0	缓冲区0传输长度

● 警告

如果 `clk_sys` 和 `clk_usb` 以不同速度运行，缓冲区控制寄存器中的 `available` 和 `stall` 位应在其它数据之后设置。否则，控制器可能会启动包含前一数据包数据的事务。也就是说，控制器可能会看到 `available` 位被设置，但数据 `pid` 或 `length` 却来自前一数据包。

4.1.2.8. 设备控制器

本节详述设备控制器在接收来自主机的各类数据包时的工作原理。

4.1.2.8.1. SETUP

设备控制器必须始终接受来自主机的 `setup` 数据包。因此，DPSRAM 的前 8 字节为 `setup` 数据包专门预留空间。

文献[2]指出，接收setup数据包同时会清除EP0上的任何stall位。因此，EP0的stall位由EP_STALL_ARM寄存器中的两个位进行门控。接收到setup数据包时，这些位将被清除。这意味着，若要在EP0上发送stall信号，必须同时设置缓冲区控制寄存器中的stall位及EP_STALL_ARM中的相应位。

在无错误的情况下，setup包将被存放于DPSRAM偏移 `0x0` 处的setup包缓冲区。设备控制器随后将发送ACK响应。

最后，将设置SIE_STATUS.SETUP_REC，以指示已接收到setup包。若程序员启用了SETUP_REC中断（详见INTE），则此操作将触发中断。

4.1.2.8.2. IN

从设备角度来看，`IN`传输指的是将数据传输进入主机。当从主机接收到`IN`令牌时，请求将按以下方式处理：

TOKEN阶段：

- 若缓冲区控制寄存器中设置了`STALL`（且对于EP0，相应的EP_STALL_ARM位已设置），则发送STALL响应并返回空闲状态。
- 如果缓冲区控制中设置了`AVAILABLE`和`FULL`位，则进入该阶段。
- 否则发送`NAK`，除非该端点为等时端点，在此情况下进入空闲状态。

数据阶段：

- 发送数据。若为等时传输，则进入空闲状态。否则进入确认(ACK)阶段。

确认(ACK)阶段：

- 等待主机发送的`ACK`包。若超时，则产生超时错误。接收到`ACK`后，数据包传输完成，转入状态阶段。

状态阶段：

- 如果这是本次传输的最后一个缓冲区（即缓冲区控制寄存器中的`LAST_BUFFER`位被设置），则设置SIE_STATUS.TRANS_COMPLETE。
- 若端点为双缓冲，则切换缓冲区选择至另一缓冲区。
- 在BUFF_STATUS中设置相应位以指示缓冲区已完成。处理该事件时，程序员应读取BUFF_CPU_SHOULD_HANDLE以判断完成的是缓冲区0还是缓冲区1。如果端点采用双缓冲，两个缓冲区均可能已完成。已清除的BUFF_STATUS位将再次被置位，并且此时BUFF_CPU_SHOULD_HANDLE将发生变化。**
- 更新缓冲区控制寄存器对应半部分的状态：`length`、`pid` 及 `last_buff` 被设置，其余字段写为零。

若向主机发送了`NAK`，主机将稍后重新尝试。

4.1.2.8.3. OUT

当从主机接收到`OUT`令牌时，按如下方式处理请求：

TOKEN阶段：

- `DATA pid` 是否与缓冲区控制寄存器中指定的值一致？若不一致，则触发SIE_STATUS.DATA_SEQ_ERROR。（Isochronous端点的数据`pid`不进行检查，因为Isochronous数据始终以`DATA0 pid`发送。）
- `AVAILABLE`位是否被设置且`FULL`位是否未设置？若是，则进入数据阶段；除非`STALL`位被置位，此时设备控制器将以`STALL`响应。

数据阶段：

- 将接收的数据存储到缓冲区。若为等时传输，则进入状态阶段。否则进入ACK阶段。

确认(ACK)阶段：

- 发送ACK，进入状态阶段。

状态阶段：

详见第4.1.2.8.2节的状态阶段。唯一区别在于缓冲区控制寄存器中的 FULL位被设置，表示数据已接收，而在 IN情况下， FU LL位则被清除，表示数据已发送。

4.1.2.8.4. 挂起与恢复

USB设备控制器支持挂起及恢复功能，并支持远程恢复（由SIE_CTRL.RESUME触发），设备可以主动发起恢复。SIE_STATUS寄存器含有一个中断/状态位。通常无需启用挂起和恢复中断，因为大多数设备无需处理挂起与恢复。

当设备未检测到主机每1毫秒发送的帧起始包时，设备进入挂起状态。

i 注意

如果启用挂起中断，则设备首次连接且总线处于空闲状态时，您很可能会收到挂起中断。在主机开始发送帧起始数据包之前，总线可能会空闲数毫秒。如果未连接VBUS检测电路，设备断开时也会产生挂起中断。这是因为缺乏VBUS检测时，无法区分设备断开与挂起状态。

4.1.2.8.5. 勘误

设备控制器存在两个硬件问题，RP2040B0和RP2040B1通过软件进行规避，RP2040B2已在硬件层面修复。详情请参阅RP2040-E2和RP2040-E5。

4.1.2.9. 主机控制器

主机控制器的设计与设备控制器类似。所有事务均由主机发起，主机始终处理其自身发起的事务。因此，只有一组端点控制／端点缓冲区控制寄存器。还有额外硬件用于在无软件控制事务进行时在后台轮询中断端点。

主机需每1毫秒向设备发送保持活动包，以防止设备进入挂起状态。在全速模式下，通过发送一个 SOF（帧开始）包实现此操作。在低速模式下，发送一个 EOP（数据包结束）信号。配置控制器时，应启用SIE_CTRL.KEEP_ALIVE_EN和SIE_CTRL.SOF_EN位以允许这些数据包。

SIE_CTRL寄存器中的若干位用于启动主机事务：

- SEND_SETUP- 发送一个setup包。该命令通常与RECEIVE_TRANS结合使用，先发送setup包，随后进行设备预期的附加数据传输。
- SEND_TRANS- 该传输方向为来自主机的 OUT
- RECEIVE_TRANS- 该传输方向为至主机的 IN
- START_TRANS - 启动传输 - 非锁存
- STOP_TRANS - 停止当前传输 - 非锁存
- PREAMBLE_ENABLE- 用于向全速集线器上的低速设备发送数据包。该设置会在主机发送的每个数据包之前发送一个 PRE E token数据包（即 pre, token, pre, data, pre, ack）。
- SOF_SYNC- SOF同步位用于延迟事务，直至下一个SOF发生后。这对于中断和同步传输端点尤为有用。主机控制器防止64字节事务与SOF数据包发生冲突。

对于较长的同步数据包，软件需负责使用SOF同步位并限制单帧内发送数据包的数量，以防止冲突。若事务中包含多个数据包，SOF同步位仅适用于第一个数据包。

● 警告

START_TRANS位与SIE_CTRL寄存器中的其他控制位分别同步。应将SIE_CTRL寄存器中的START_TRANS位与其他数据分开设置，以确保在控制器被触发开始传输时，寄存器内容保持稳定。这是必要的，因为处理器时钟clk_sys可能与clk_usb时钟异步。

- 写入 SIE_CTRL 中除 START_TRANS 以外的字段
- 执行nop操作，持续若干clk_sys周期，以确保至少经过两个clk_usb周期。例如，如果clk_sys运行于125MHz，而clk_usb运行于48MHz，则125/48向上取整为6，即6条nop指令。
- 设置START_TRANS位。

4.1.2.9.1. SETUP

从主机发送的 SETUP 数据包始终来自DPSRAM中偏移地址 0x0 处专用的8字节空间。

与设备控制器类似，没有与 SETUP 数据包相关联的控制寄存器。这些参数为硬编码形式，并在您写入且设置了 SEND_SETUP 位的 START_TRANS 时加载到硬件中。一旦设置包发送完成，主机状态机将等待设备返回 ACK。若发生超时，则会触发 RX_TIMEOUT 错误。若设置了 SEND_TRANS 位，主机状态机将进入 OUT 阶段。通常，SEND_SETUP 包与 RECEIVE_TRANS 位配合使用，因此发送设置包后将进入 IN 阶段。

4.1.2.9.2. IN

当设置了 START_TRANS 位与 RECEIVE_TRANS 位时，将触发一次 IN 传输。若先前设置了 SEND_SETUP 位，可能会发送一个 SETUP 包。

控制阶段：

- 读取位于 0x80 的 EPx control 寄存器以获取端点信息：
 - 是否启用双缓冲？
 - 需启用的中断
 - 数据缓冲区基地址，或双缓冲模式下的数据缓冲区
 - 端点类型
- 读取位于 0x100 的 EPx 缓冲区控制 寄存器以获取端点缓冲区信息，例如传输长度和数据 PID。主机状态机仍会检查 AVAILABLE 位的存在，因此该位须设为 1，且 FULL 位须清零。事务仅在满足此条件时发生。

TOKEN阶段：

- 向设备发送 IN 令牌包。目标设备地址和端点取自 ADDR_ENDP 寄存器。

数据阶段：

- 接收来自设备的首个数据包。若设备无响应，触发 RX 超时错误。如数据包的数据 PID 错误，触发数据序列错误。

确认(ACK)阶段：

- 向设备发送 ACK 信号

状态阶段：

- 设置BUFF_STATUS位并更新缓冲区控制寄存器。将设置 FULL、 LAST_BUFF（如适用）、 DATA_PID及 WR_LEN。若为传输中的最后一个缓冲区，将设置TRANS_COMPLETE。

控制阶段（第2部分）：

- 主机状态机将持续执行 IN事务，直到在buffer_control寄存器中检测到 LAST_BUFF。若主机处于双缓冲模式，主机控制器将在buffer_control寄存器的 BUFO和 BUF1部分之间切换。否则，将持续读取缓冲区控制寄存器的缓冲区0，等待 FULL标志清除且 AVAILABLE标志设置后，再启动下一次IN事务（即在控制阶段等待）。设备可向主机发送零长度数据包，以示其无更多数据。此时，无论 LAST_BUFF标志是否设置，主机状态机均停止接收更多数据。主机软件可通过检测缓冲区控制寄存器中数据长度为0且 BUFF_DONE标志被设置来确认此情况。

● 警告

USB主机控制器存在一个漏洞（RP2040-E4），导致写回缓冲区控制寄存器的状态可能出现在寄存器的错误半区。位0至15用于缓冲区0，位16至31用于缓冲区1。主机控制器设有缓冲区选择器，每次传输完成后会切换该选择器。即使在单缓冲模式下，写回状态信息至缓冲区控制寄存器时，该缓冲区选择器仍被错误使用。读取缓冲区控制寄存器时不使用缓冲区选择器。由此可见，主机软件需跟踪缓冲区选择器状态，若缓冲区选择器为1，则应将缓冲区控制寄存器右移16位。

详细信息请参阅 RP2040-E4。

4.1.2.9.3. OUT

当 START_TRANS位置位且 SEND_TRANS位被设置时，将触发一个OUT传输。若 SEND_SETUP位已设置，则之前可能会发送一个 SETUP包。

控制阶段：

- 读取EPx control以获取端点信息（同4.1.2.9.2节）。
- 读取EPx缓冲区控制以获取传输长度和数据pid。 AVAILABLE和 FULL必须被设置，传输方可开始。

TOKEN阶段

- 向设备发送 OUT数据包。目标设备地址和端点取自ADDR_ENDP寄存器。

数据阶段：

- 向设备发送首个数据包。若端点类型为等时传输，则无ACK阶段，主控制器将直接进入状态阶段。若收到 ACK，则进入状态阶段。否则：
 - 若未收到回复，则触发SIE_STATUS.RX_TIMEOUT。
 - 若收到 NAK，则触发SIE_STATUS.NAK_REC并重新发送数据包。
 - 若收到 STALL，则触发SIE_STATUS.STALL_REC并进入空闲状态。

状态阶段：

- 设置BUFF_STATUS位并更新缓冲区控制寄存器， FULL位将被置为0。若为传输中的最后一个缓冲区，将设置TRANS_COMPLETE。

● 警告

上述IN部分所述的错误（RP2040-E4）同样适用于OUT部分。

控制阶段（第2部分）：

若此次传输尚未结束，则等待 FULL和 AVAILABLE在EPx缓冲区控制寄存器中再次被设置。

4.1.2.9.4. 中断端点

主机控制器可以轮询多个设备上的中断端点（最大为15个端点）。要启用这些功能，程序员必须：

- 在主机控制器上选择下一个空闲的中断端点槽（从1开始，最多15个）
- 如同普通的 IN或 OUT传输一般，配置相应的端点控制寄存器和缓冲区控制寄存器。注意，中断端点仅支持单缓冲，因此缓冲区控制寄存器中的BUF1部分无效。
- 在对应的 ADDR_ENDP 寄存器（ADDR_ENDP1至ADDR_ENDP15）中设置设备地址及端点号。
若设备为低速且接入全速集线器，需设置前导码位并同时设置端点方向位。
- 在INT_EP_CTRL寄存器中设置中断端点激活位（即寄存器的第1至第15位）。

通常，中断端点为 IN方向传输。例如，USB集线器会被轮询以检查其任一端口的状态是否发生变化。如无变化，集线器将向控制器回复 NAK，且无任何操作发生。

同理，除非鼠标自上次轮询中断端点以来移动过，否则鼠标将回复 NAK。

中断端点由控制器在主机控制器发送完一个 SOF数据包后进行轮询。

控制器循环遍历1至15，尝试轮询在INT_EP_CTRL寄存器中EP_ACTIVE位为1的所有中断端点。随后，控制器读取端点控制寄存器及缓冲控制寄存器，以确认是否存在可用缓冲区（即，对于OUT传输，应为FULL且AVAILABLE；对于IN传输，应为 NOT FULL且AVAILABLE）。如无可用缓冲区，控制器将继续轮询下一个中断端点槽。

若存在可用缓冲区，传输将按正常的 IN或 OUT传输处理，当中断端点拥有有效缓冲区时， BUFF_DONE标志将在BUFF_STATUS寄存器中被置位。BUF_CPU_SHOULD_HANDLE 对中断端点无效，因为仅能完成单个缓冲区（RP2040-E3）。

4.1.2.10. VBUS 控制

USB控制器可连接至 GPIO 引脚（详见第 2.19 节），以实现 VBUS 控制：

- VBUS 使能，用于在主机模式下启用 VBUS。VBUS 使能位于 SIE_CTRL 寄存器中。
- VBUS 检测，用于在设备模式下检测 VBUS 的存在。VBUS 检测为 SIE_STATUS 寄存器中的一位，并可触发 VBUS_DETECT 中断（由 INTE 使能）。
- VBUS 过流，用于检测过流事件。适用于设备和主机两种模式。VBUS 过流为 SIE_STATUS 寄存器中的一位。

无须将上述引脚连接至 GPIO。主机可持续供电 VBUS，并通过 DP 或 DM 引脚电平拉高以检测设备连接。VBUS检测可在 USB_PWR中被强制执行。

4.1.3. 程序员模型

4.1.3.1. TinyUSB

RP2040 TinyUSB端口应视为该USB控制器的参考实现。该端口可在以下位置找到：

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/dcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/hcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/rp2040_usb.h

4.1.3.2. 独立设备示例

独立USB设备示例`dev_lowlevel`，有助于理解如何与USB控制器交互，而无需理解TinyUSB抽象层。除了端点0外，独立设备还具有两个批量传输端点：EP1 OUT和EP2 IN。该设备设计为将从EP1接收的所有数据发送至EP2。示例附带一个简易Python脚本，将“Hello World”写入EP1，并验证该数据是否被正确接收到EP2。本节代码将引导您设置USB设备控制器，以接收设置包并响应该设置包。

图58。dev_lowlevel USB设备示例的USB分析仪跟踪。控制传输即设备枚举。第一个批量OUT（从主机发出）传输，蓝色高亮部分，表示主机向设备发送“Hello World”。第二个批量IN（传入主机）传输，表示设备将“Hello World”返回给主机。

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors
0	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
1	S	SET	0	0	SET_ADDRESS	New address 7	0x0000	
2	S	GET	7	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
3	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor
4	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	4 Descriptors
5	S	GET	7	0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	Lang Supported
6	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	Pico Test Device
7	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 1	Language ID 0x0409	Raspberry Pi
8	S	SET	7	0	SET_CONFIGURATION	New Configuration 1	0x0000	
9	S	Bulk	ADDR	ENDP	Bytes Transferred			
10	S	OUT	7	1		12		
		Bulk	ADDR	ENDP	Bytes Transferred			
		IN	7	2		12		

4.1.3.2.1. 设备控制器初始化

以下代码用于初始化USB设备。

Pico示例代码：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第183至217行

```

183 void usb_device_init() {
184     //重置USB控制器
185     reset_unreset_block_num_wait_blocking(RESET_USBCTRL);
186
187     //清除dpram中的任何先前状态
188     memset(usb_dpram, 0, sizeof(*usb_dpram)); ①
189
190     //在处理器上启用USB中断
191     irq_set_enabled(USBCTRL_IRQ, true);
192
193     //将控制器复用于板载USB物理层

```

```

194     usb_hw->muxing = USB_USB_MUXING_TO_PHY_BITS | USB_USB_MUXING_SOFTCON_BITS;
195
196     // 强制VBUS检测，使设备识别为已连接主机
197     usb_hw->pwr = USB_USB_PWR_VBUS_DETECT_BITS | USB_USB_PWR_VBUS_DETECT_OVERRIDE_EN_BITS;
198
199     // 以设备模式启用USB控制器
200     usb_hw->main_ctrl = USB_MAIN_CTRL_CONTROLLER_EN_BITS;
201
202     // 使能每个EP0事务的中断
203     usb_hw->sie_ctrl = USB_SIE_CTRL_EP0_INT_1BUF_BITS; ②
204
205     // 启用缓冲区完成、总线复位
206     // 以及接收setup包时的中断
207     usb_hw->inte = USB_INTS_BUFF_STATUS_BITS |
208                     USB_INTS_BUS_RESET_BITS |
209                     USB_INTS_SETUP_REQ_BITS;
210
211     // 设置端点（端点控制寄存器）
212     // 由设备配置描述
213     usb_setup_endpoints();
214
215     // 通过在DP线上启用上拉，激活全速设备
216     usb_hw_set->sie_ctrl = USB_SIE_CTRL_PULLUP_EN_BITS;
217 }

```

4.1.3.2.2. 配置EP1和EP2的端点控制寄存器

函数`usb_configure_endpoints`循环遍历设备配置中定义的每个端点（包括未定义端点控制寄存器的EP0输入和EP0输出），并调用`usb_configure_endpoint`函数。此操作设置该端点的端点控制寄存器：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第149至164行

```

149 void usb_setup_endpoint(const struct usb_endpoint_configuration *ep) {
150     printf("设置端点 0x%02x, 缓冲区地址 0x%p\n", ep->descriptor->bEndpointAddress, ep->data_buffer); 151
152
153     // EP0 不具备此项，如是则返回
154     if (!ep->endpoint_control) {
155         return;
156     }
157
158     // 获取 USB 控制器 DPRAM 中的缓冲区偏移
159     uint32_t dpram_offset = usb_buffer_offset(ep->data_buffer);
160     uint32_t reg = EP_CTRL_ENABLE_BITS
161             | EP_CTRL_INTERRUPT_PER_BUFFER
162             | (ep->descriptor->bmAttributes << EP_CTRL_BUFFER_TYPE_LSB)
163             | dpram_offset;
164     *ep->endpoint_control = reg;
165 }

```

4.1.3.2.3. 接收设置包

接收到设置包时会触发中断，故中断处理程序必须处理此事件：

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第494至504行

```

494 void isr_usbctrl(void) {
495     // USB 中断处理函数
496     uint32_t status = usb_hw->ints;
497     uint32_t handled = 0;
498
499     // 设置数据包已接收
500     if (status & USB_INTS_SETUP_REQ_BITS) {
501         handled |= USB_INTS_SETUP_REQ_BITS;
502         usb_hw_clear->sie_status = USB_SIE_STATUS_SETUP_REC_BITS;
503         usb_handle_setup_packet();
504     }

```

设置数据包被写入 USB RAM 的前 8 个字节，故设置数据包处理函数将该内存区域转换为 `struct usb_setup_packet *` 类型。

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第 383-427 行

```

383 void usb_handle_setup_packet(void) {
384     volatile struct usb_setup_packet *pkt = (volatile struct usb_setup_packet *) &usb_dpram
385             ->setup_packet;
386     uint8_t req_direction = pkt->bmRequestType;
387     uint8_t req = pkt->bRequest;
388
389     // 将 EP0 IN 的 PID 重置为 1
390     usb_get_endpoint_configuration(EP0_IN_ADDR)->next_pid = 1u;
391
392     if (req_direction == USB_DIR_OUT) {
393         if (req == USB_REQUEST_SET_ADDRESS) {
394             usb_set_device_address(pkt);
395         } else if (req == USB_REQUEST_SET_CONFIGURATION) {
396             usb_set_device_configuration(pkt);
397         } else {
398             usb_acknowledge_out_request();
399             printf("其他 OUT 请求 (0x%x)\r\n", pkt->bRequest);
400         }
401     } else if (req_direction == USB_DIR_IN) {
402         if (req == USB_REQUEST_GET_DESCRIPTOR) {
403             uint16_t descriptor_type = pkt->wValue >> 8;
404
405             switch (descriptor_type) {
406                 case USB_DT_DEVICE:
407                     usb_handle_device_descriptor(pkt);
408                     printf("GET DEVICE DESCRIPTOR\r\n");
409                     break;
410
411                 case USB_DT_CONFIG:
412                     usb_handle_config_descriptor(pkt);
413                     printf("GET CONFIG DESCRIPTOR\r\n");
414                     break;
415
416                 case USB_DT_STRING:
417                     usb_handle_string_descriptor(pkt);
418                     printf("GET STRING DESCRIPTOR\r\n");
419                     break;
420
421                 default:
422                     printf("未处理的 GET_DESCRIPTOR 类型 0x%x\r\n", descriptor_type);
423             }
424         } else {
425             printf("其他 IN 请求 (0x%x)\r\n", pkt->bRequest);
426         }
427     }

```

```

425      }
426  }
427 }
```

4.1.3.2.4 在 EP0 IN 端点响应设置包

主机首先请求设备描述符，以下代码处理该设置请求。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第266至273行

```

266 void usb_handle_device_descriptor(volatile struct usb_setup_packet *pkt) {
267     const struct usb_device_descriptor *d = dev_config.device_descriptor;
268     // EP0 输入端点
269     struct usb_endpoint_configuration *ep = usb_get_endpoint_configuration(EP0_IN_ADDR);
270     // 始终响应 PID 1
271     ep->next_pid = 1;
272     usb_start_transfer(ep, (uint8_t *) d, MIN(sizeof(struct usb_device_descriptor), pkt-
>wLength));
273 }
```

`usb_start_transfer` 函数将要发送的数据复制至相应的硬件缓冲区，并配置缓冲区控制寄存器。缓冲区控制寄存器一旦被写入，设备控制器将向主机发送响应数据。在此之前，设备将回复NAK。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第238-260行

```

238 void usb_start_transfer(struct usb_endpoint_configuration *ep, uint8_t *buf, uint16_t len) {
239     // 我们断言长度不超过64字节，以简化示例。
240     // 有关多包传输，请参见 tinyusb 端口。
241     assert(len <= 64);
242
243     printf("开始传输，长度 %d，端点地址 0x%x\n", len, ep->descriptor->bEndpointAddress);
244
245     // 准备缓冲区控制寄存器的值
246     uint32_t val = len | USB_BUF_CTRL_AVAIL;
247
248     if (ep_is_tx(ep)) {
249         // 需将数据从用户缓冲区复制至USB内存
250         memcpy((void *) ep->data_buffer, (void *) buf, len);
251         // 标记为已满
252         val |= USB_BUF_CTRL_FULL;
253     }
254
255     // 设置pid并翻转，以便进行下一次传输
256     val |= ep->next_pid ? USB_BUF_CTRL_DATA1_PID : USB_BUF_CTRL_DATA0_PID;
257     ep->next_pid ^= 1u;
258
259     *ep->buffer_control = val;
260 }
```

4.1.4. 寄存器列表

USB寄存器起始基址为 **0x50110000**（在SDK中定义为USBCTRL_REGS_BASE）。

表397. USB
寄存器列表

偏移量	名称	说明
0x00	ADDR_ENDP	设备地址及端点控制
0x04	ADDR_ENDP1	中断端点1。仅适用于HOST模式。
0x08	ADDR_ENDP2	中断端点2。仅适用于HOST模式。
0x0c	ADDR_ENDP3	中断端点3。仅适用于HOST模式。
0x10	ADDR_ENDP4	中断端点4。仅适用于HOST模式。
0x14	ADDR_ENDP5	中断端点5。仅适用于HOST模式。
0x18	ADDR_ENDP6	中断端点6。仅适用于HOST模式。
0x1c	ADDR_ENDP7	中断端点7。仅适用于HOST模式。
0x20	ADDR_ENDP8	中断端点8。仅适用于HOST模式。
0x24	ADDR_ENDP9	中断端点9。仅适用于HOST模式。
0x28	ADDR_ENDP10	中断端点10。仅适用于HOST模式。
0x2c	ADDR_ENDP11	中断端点11。仅适用于HOST模式。
0x30	ADDR_ENDP12	中断端点12。仅适用于HOST模式。
0x34	ADDR_ENDP13	中断端点13。仅适用于HOST模式。
0x38	ADDR_ENDP14	中断端点14。仅适用于HOST模式。
0x3c	ADDR_ENDP15	中断端点15。仅适用于HOST模式。
0x40	MAIN_CTRL	主控制寄存器
0x44	SOF_WR	设置主控制器中的 SOF（帧开始）帧编号。 SOF 数据包每 1 毫秒发送一次，主机每次将帧编号递增 1。
0x48	SOF_RD	读取最近接收到的 SOF（帧开始）帧编号。设备模式下，最后接收到的来自主机的 SOF 帧。主机模式下，最后由主机发送的 SOF 帧。
0x4c	SIE_CTRL	SIE 控制寄存器
0x50	SIE_STATUS	SIE 状态寄存器
0x54	INT_EP_CTRL	中断端点控制寄存器
0x58	BUFF_STATUS	缓冲区状态寄存器。此位置位表示端点上的缓冲区已完成（前提是缓冲区中断已启用）。有可能两个缓冲区同时完成，因此清除缓冲区状态位后，下一时钟周期可能会立即重新置位。
0x5c	BUFF_CPU_SHOULD_HANDLE	应处理的双缓冲区中的哪一个。仅在每缓冲区产生中断时有效（即非每两个缓冲区）。对主机中断端点轮询无效，因为它们仅为单缓冲区。
0x60	EP_ABORT	仅限设备：可设置为忽略该端点的缓冲区控制寄存器，以便撤销缓冲区。在清除此位之前，每次访问该端点时均会发送NAK。当可安全修改缓冲区控制寄存器时，EP_ABORT_DONE中对应位将被置位。

偏移量	名称	说明
0x64	EP_ABORT_DONE	仅限设备：与 EP_ABORT 配合使用。端点空闲时设置此位，以通知程序员可安全修改缓冲区控制寄存器。
0x68	EP_STALL_ARM	设备：此位须与缓冲区控制寄存器中的 STALL 位同时设置，以便在 EP0 端点发送 STALL。设备控制器在收到 SETUP 包时会清除此位，因为 USB 规范要求在收到 SETUP 包时清除 STALL 状态。
0x6c	NAK_POLL	由主机控制器使用。设置设备响应 NAK 时重试前的等待时间，单位为微秒。
0x70	EP_STATUS_STALL_NAK	设备：当 IRQ_ON_NAK 或 IRQ_ON_STALL 位被置位时，此位亦被置位。对于 EP0，该信号由 SIE_CTRL 提供。对于所有其他端点，其来源于端点控制寄存器。
0x74	USB_MUXING	USB 控制器的连接位置。默认应为 to_phy。
0x78	USB_PWR	当 VBUS 信号未连接至 GPIO 时，用于电源信号的覆盖。先设置覆盖值，再启用覆盖以切换至该值。
0x7c	USBPHY_DIRECT	此寄存器允许直接控制 USB 物理层。需配合 usbphy_direct_override 寄存器使用以启用各覆盖位。
0x80	USBPHY_DIRECT_OVERRIDE	usbphy_direct 中各控制项的覆盖使能。
0x84	USBPHY_TRIM	用于调整 USB 物理层下拉电阻的调节值。
0x8c	INTR	原始中断
0x90	INTE	中断使能
0x94	INTF	中断强制
0x98	INTS	掩码及强制后的中断状态

USB: ADDR_ENDP 寄存器

偏移: 0x00

描述

设备地址及端点控制

表 398。
ADDR_ENDP 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:16	ENDPOINT : 设备端点，用于发送数据。仅在主机模式下有效。	读写	0x0
15:7	保留。	-	-
6:0	ADDRESS : 设备模式下设备应响应的地址，根据主机发送的 SET_ADDR 请求包设置。主机模式下设置为待通信设备的地址。	读写	0x00

USB: ADDR_ENDP1、ADDR_ENDP2、...、ADDR_ENDP14、ADDR_ENDP15 寄存器

偏移: 0x04, 0x08, ..., 0x38, 0x3c

描述

中断端点 N。仅适用于HOST模式。

表399。
ADDR_ENDP1,
ADDR_ENDP2, ...,
ADDR_ENDP14,
ADDR_ENDP15
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	INTEP_PREAMBLE : 中断端点需带前导码 (在全速集线器上的低速设备)	读写	0x0
25	INTEP_DIR : 中断端点方向。输入=0, 输出=1	读写	0x0
24:20	保留。	-	-
19:16	ENDPOINT : 中断端点编号	读写	0x0
15:7	保留。	-	-
6:0	ADDRESS : 设备地址	读写	0x00

USB: MAIN_CTRL 寄存器

偏移: 0x40

描述

主控制寄存器

表400。
MAIN_CTRL 寄存器

位	描述	类型	复位值
31	SIM_TIMING : 仿真用缩减时序	读写	0x0
30:2	保留。	-	-
1	HOST_NDEVICE : 设备模式=0, 主机模式=1	读写	0x0
0	CONTROLLER_EN : 启用控制器	读写	0x0

USB: SOF_WR 寄存器

偏移: 0x44

描述

设置主机控制器中的SOF (帧起始) 帧编号。SOF 数据包每 1 毫秒发送一次，主机每次将帧编号递增 1。

表 401. SOF_WR
寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10:0	计数	WF	0x000

USB: SOF_RD 寄存器

偏移: 0x48

描述

读取最近接收到的 SOF (帧开始) 帧编号。设备模式下，最后接收到的来自主机的 SOF 帧。主机模式下，最后由主机发送的 SOF 帧。

表 402. SOF_RD
寄存器

位	描述	类型	复位值
31:11	保留。	-	-

位	描述	类型	复位值
10:0	计数	只读	0x000

USB：SIE_CTRL 寄存器

偏移：0x4c

描述

SIE 控制寄存器

表 403. SIE_CTRL 寄存器

位	描述	类型	复位值
31	EPO_INT_STALL : 设备：当 EP0 发送 STALL 时，设置 EP_STATUS_STALL_NAK 中相应位	读写	0x0
30	EPO_DOUBLE_BUF : 设备：EP0 单缓冲区 = 0，双缓冲区 = 1	读写	0x0
29	EPO_INT_1BUF : 设备：每完成 EP0 单个缓冲区时，设置 BUFF_STATUS 中相 应位	读写	0x0
28	EPO_INT_2BUF : 设备：每完成 EP0 两个缓冲区时，设置 BUFF_STATUS 中相 应位	读写	0x0
27	EPO_INT_NAK : 设备：当 EP0 发送 NAK 时，设置 EP_STATUS_STALL_NAK 中相 应位	读写	0x0
26	DIRECT_EN : 直接总线驱动使能	读写	0x0
25	DIRECT_DP : 直接控制 DP	读写	0x0
24	DIRECT_DM : 直接控制 DM	读写	0x0
23:19	保留。	-	-
18	TRANSCEIVER_PD : 断电总线收发器	读写	0x0
17	RPU_OPT : 设备：上拉电阻强度 (0=1K2, 1=2k3)	读写	0x0
16	PULLUP_EN : 设备：启用上拉电阻	读写	0x0
15	PULLDOWN_EN : 主机：启用下拉电阻	读写	0x0
14	保留。	-	-
13	RESET_BUS : 主机：复位总线	SC	0x0
12	RESUME : 设备：远程唤醒设备可在挂起后自发恢复。	SC	0x0
11	VBUS_EN : 主机：启用 VBUS	读写	0x0
10	KEEP_ALIVE_EN : 主机：启用保持活动包（针对低速总线）	读写	0x0
9	SOF_EN : 主机：启用 SOF 生成（针对全速总线）	读写	0x0
8	SOF_SYNC : 主机：延迟数据包至 SOF 后	读写	0x0
7	保留。	-	-
6	PREAMBLE_EN : 主机：为全速集线器上的低速设备启用前导码	读写	0x0
5	保留。	-	-
4	STOP_TRANS : 主机：停止事务	SC	0x0
3	RECEIVE_DATA : 主机：接收事务 (IN 至主机)	读写	0x0
2	SEND_DATA : 主机：发送事务 (OUT 从主机)	读写	0x0

位	描述	类型	复位值
1	SEND_SETUP : 主机: 发送设置包	读写	0x0
0	START_TRANS : 主机: 启动事务	SC	0x0

USB: SIE_STATUS 寄存器

偏移: 0x50

描述

SIE 状态寄存器

表 404。
SIE_STATUS 寄存器

位	描述	类型	复位值
31	DATA_SEQ_ERROR : 数据序列错误。 设备在以下情况下可能触发序列错误： * 接收到一个 SETUP 包后紧跟 DATA1 包 (数据阶段应始终为 DATA0) * 从主机接收到的 OUT 包其数据 PID 与从 DPRAM 读取的缓冲区控制寄存器中的数据 PID 不匹配 主机在以下情况下可能触发数据序列错误： * 设备发送的 IN 包数据 PID 错误	WC	0x0
30	ACK_REC : 已接收 ACK。由主机和设备共同触发。	WC	0x0
29	STALL_REC : 主机: 已接收 STALL	WC	0x0
28	NAK_REC : 主机: 已接收 NAK	WC	0x0
27	RX_TIMEOUT : 若在 USB 规范规定的最长时间内未接收到 ACK, 主机和设备均会触发接收超时。	WC	0x0
26	RX_OVERFLOW : 当接收数据速率过快时, 串行接收引擎触发接收溢出。	WC	0x0
25	BIT_STUFF_ERROR : 位填充错误。由串行接收引擎触发。	WC	0x0
24	CRC_ERROR : CRC 校验错误。由串行接收引擎触发。	WC	0x0
23:20	保留。	-	-
19	BUS_RESET : 设备: 接收到总线复位信号	WC	0x0
18	TRANS_COMPLETE : 事务完成。 由设备触发, 条件如下： * 当 IN 包或 OUT 包发送时, 缓冲区控制寄存器中的 LAST_BUFF 位被设置 由主机触发, 条件如下： * 发送设置包后未跟随数据输入或数据输出事务 * 接收到 IN 包且缓冲区控制寄存器中的 LAST_BUFF 位被设置 * 收到长度为零的 IN 包 * 发送 OUT 包且缓冲区控制寄存器中的 LAST_BUFF 位被设置	WC	0x0
17	SETUP_REC : 设备: 收到设置包	WC	0x0

位	描述	类型	复位值
16	CONNECTED : 设备: 已连接	WC	0x0
15:12	保留。	-	-
11	RESUME : 主机: 设备已发起远程恢复。设备: 主机已发起恢复。	WC	0x0
10	VBUS_OVER_CURR : 检测到VBUS过流	只读	0x0
9:8	SPEED : 主机: 设备速度。断开连接 = 00, 低速 = 01, 全速 = 10	WC	0x0
7:5	保留。	-	-
4	SUSPENDED : 总线处于挂起状态。适用于设备和主机。若未启用Keep Alive或SOF帧，则主机和设备将进入挂起状态。	WC	0x0
3:2	LINE_STATE : USB总线线路状态	只读	0x0
1	保留。	-	-
0	VBUS_DETECTED : 设备: 检测到VBUS	只读	0x0

USB: INT_EP_CTRL寄存器

偏移: 0x54

说明

中断端点控制寄存器

表405。
INT_EP_CTRL寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:1	INT_EP_ACTIVE : 主机: 启用中断端点1 → 15	读写	0x0000
0	保留。	-	-

USB: BUFF_STATUS寄存器

偏移: 0x58

说明

缓冲区状态寄存器。此位置位表示端点上的缓冲区已完成（前提是缓冲区中断已启用）。有可能两个缓冲区同时完成，因此清除缓冲区状态位后，下一时钟周期可能会立即重新置位。

表406。
BUFF_STATUS
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0

位	描述	类型	复位值
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB：BUFF_CPU_SHOULD_HANDLE寄存器

偏移：0x5c

描述

应处理的双缓冲区中的哪一个。仅在每缓冲区产生中断时有效（即非每两个缓冲区）。对主机中断端点轮询无效，因为它们仅为单缓冲区。

表407。
BUFF_CPU_SHOULD_HANDLE寄存器

位	描述	类型	复位值
31	EP15_OUT	只读	0x0
30	EP15_IN	只读	0x0
29	EP14_OUT	只读	0x0
28	EP14_IN	只读	0x0
27	EP13_OUT	只读	0x0
26	EP13_IN	只读	0x0

位	描述	类型	复位值
25	EP12_OUT	只读	0x0
24	EP12_IN	只读	0x0
23	EP11_OUT	只读	0x0
22	EP11_IN	只读	0x0
21	EP10_OUT	只读	0x0
20	EP10_IN	只读	0x0
19	EP9_OUT	只读	0x0
18	EP9_IN	只读	0x0
17	EP8_OUT	只读	0x0
16	EP8_IN	只读	0x0
15	EP7_OUT	只读	0x0
14	EP7_IN	只读	0x0
13	EP6_OUT	只读	0x0
12	EP6_IN	只读	0x0
11	EP5_OUT	只读	0x0
10	EP5_IN	只读	0x0
9	EP4_OUT	只读	0x0
8	EP4_IN	只读	0x0
7	EP3_OUT	只读	0x0
6	EP3_IN	只读	0x0
5	EP2_OUT	只读	0x0
4	EP2_IN	只读	0x0
3	EP1_OUT	只读	0x0
2	EP1_IN	只读	0x0
1	EP0_OUT	只读	0x0
0	EP0_IN	只读	0x0

USB: EP_ABORT 寄存器

偏移: 0x60

描述

仅限设备：可设置为忽略该端点的缓冲区控制寄存器，以便撤销缓冲区。在清除此位之前，每次访问该端点时均会发送NAK。当可以安全修改缓冲区控制寄存器时，EP_ABORT_DONE的对应位将被置位。

表 408. EP_ABORT 寄存器

位	描述	类型	复位值
31	EP15_OUT	读写	0x0
30	EP15_IN	读写	0x0
29	EP14_OUT	读写	0x0

位	描述	类型	复位值
28	EP14_IN	读写	0x0
27	EP13_OUT	读写	0x0
26	EP13_IN	读写	0x0
25	EP12_OUT	读写	0x0
24	EP12_IN	读写	0x0
23	EP11_OUT	读写	0x0
22	EP11_IN	读写	0x0
21	EP10_OUT	读写	0x0
20	EP10_IN	读写	0x0
19	EP9_OUT	读写	0x0
18	EP9_IN	读写	0x0
17	EP8_OUT	读写	0x0
16	EP8_IN	读写	0x0
15	EP7_OUT	读写	0x0
14	EP7_IN	读写	0x0
13	EP6_OUT	读写	0x0
12	EP6_IN	读写	0x0
11	EP5_OUT	读写	0x0
10	EP5_IN	读写	0x0
9	EP4_OUT	读写	0x0
8	EP4_IN	读写	0x0
7	EP3_OUT	读写	0x0
6	EP3_IN	读写	0x0
5	EP2_OUT	读写	0x0
4	EP2_IN	读写	0x0
3	EP1_OUT	读写	0x0
2	EP1_IN	读写	0x0
1	EP0_OUT	读写	0x0
0	EP0_IN	读写	0x0

USB：EP_ABORT_DONE 寄存器

偏移：0x64

描述

仅限设备：与 EP_ABORT 配合使用。端点空闲时设置此位，以通知程序员可安全修改缓冲区控制寄存器。

表 409。
EP_ABORT_DONE
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: EP_STALL_ARM 寄存器

偏移量: 0x68

描述

设备：此位必须与缓冲区控制寄存器中的 **STALL** 位同时设置，以便在 EP0 上发送 STALL 信号。

当接收到 SETUP 数据包时，设备控制器将清除这些位，因为 USB 规范要求在接收到 SETUP 数据包时必须清除 STALL 状态。

表 410。
EP_STALL_ARM
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	EP0_OUT	读写	0x0
0	EP0_IN	读写	0x0

USB：NAK_POLL 寄存器

偏移: 0x6c

描述

由主机控制器使用。设置设备响应 NAK 时重试前的等待时间，单位为微秒。

表 411. NAK_POLL
寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25:16	DELAY_FS ：全速设备的 NAK 轮询间隔	读写	0x010
15:10	保留。	-	-
9:0	DELAY_LS ：低速设备的 NAK 轮询间隔	读写	0x010

USB：EP_STATUS_STALL_NAK 寄存器

偏移: 0x70

描述

设备：当 **IRQ_ON_NAK** 或 **IRQ_ON_STALL** 位被置位时，此位亦被置位。对于 EP0，该信号由 **SIE_CTRL** 提供。对于所有其他端点，其来源于端点控制寄存器。

表 412。
EP_STATUS_STALL_NAK
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0

位	描述	类型	复位值
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: USB_MUXING 寄存器

偏移: 0x74

描述

USB 控制器的连接位置。默认应为 to_phy。

表 413。
USB_MUXING 寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	SOFTCON	读写	0x0
2	TO_DIGITAL_PAD	读写	0x0
1	TO_EXTPHY	读写	0x0
0	TO_PHY	读写	0x0

USB: USB_PWR 寄存器

偏移: 0x78

描述

当 VBUS 信号未连接至 GPIO 时，用于电源信号的覆盖。先设置覆盖值，再启用覆盖以切换至该值。

表 414。USB_PWR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-

位	描述	类型	复位值
5	OVERCURR_DETECT_EN	读写	0x0
4	OVERCURR_DETECT	读写	0x0
3	VBUS_DETECT_OVERRIDE_EN	读写	0x0
2	VBUS_DETECT	读写	0x0
1	VBUS_EN_OVERRIDE_EN	读写	0x0
0	VBUS_EN	读写	0x0

USB: USBPHY_DIRECT 寄存器

偏移: 0x7C

描述

此寄存器允许直接控制 USB 物理层。需配合 usbphy_direct_override 寄存器使用以启用各覆盖位。

表 415。
USBPHY_DIRECT
寄存器

位	描述	类型	复位值
31:23	保留。	-	-
22	DM_OVV : DM 过压	只读	0x0
21	DP_OVV : DP 过压	只读	0x0
20	DM_OVCN : DM 过流	只读	0x0
19	DP_OVCN : DP 过流	只读	0x0
18	RX_DM : DPM 引脚状态	只读	0x0
17	RX_DP : DPP 引脚状态	只读	0x0
16	RX_DD : 差分接收	只读	0x0
15	TX_DIFFMODE : TX_DIFFMODE=0: 单端模式 TX_DIFFMODE=1: 差分驱动模式 (忽略 TX_DM 和 TX_DM_OE)	读写	0x0
14	TX_FSSLEW : TX_FSSLEW=0: 低速转换速率 TX_FSSLEW=1: 全速转换速率	读写	0x0
13	TX_PD : TX 电源关闭覆盖 (如启用覆盖)。1 = 电源关闭。	读写	0x0
12	RX_PD : RX 电源关闭覆盖 (如启用覆盖)。1 = 电源关闭。	读写	0x0
11	TX_DM : 输出数据。TX_DIFFMODE=1 时忽略。 TX_DIFFMODE=0 时, 仅驱动 DPM。TX_DM_OE=1 用于启用驱动。 DPM=TX_DM	读写	0x0
10	TX_DP : 输出数据。当 TX_DIFFMODE=1 时, 驱动 DPP/DPM 差分对。 TX_DP_OE=1 用于启用驱动。DPP=TX_DP, DPM=~TX_DP 如果 TX_DIFFMODE=0, 仅驱动 DPP。TX_DP_OE=1 以启用驱动。 DPP=TX_DP	读写	0x0
9	TX_DM_OE : 输出使能。如果 TX_DIFFMODE=1, 则忽略。 如果 TX_DIFFMODE=0, 仅对 DPM 使能 OE。0 - DPM 处于高阻态; 1 - DPM 驱动中。	读写	0x0
8	TX_DP_OE : 输出使能。如果 TX_DIFFMODE=1, 对 DPP/DPM 差分对使能 OE。 0 - DPP/DPM 处于高阻态; 1 - DPP/DPM 驱动中。 如果 TX_DIFFMODE=0, 仅对 DPP 使能 OE。0 - DPP 处于高阻态; 1 - DPP 驱动中。	读写	0x0
7	保留。	-	-

位	描述	类型	复位值
6	DM_PULLDN_EN : DM 下拉使能。	读写	0x0
5	DM_PULLUP_EN : DM 上拉使能。	读写	0x0
4	DM_PULLUP_HISEL : 启用第二个 DM 上拉电阻。0 - 上拉为 Rpu2; 1 - 上拉为 Rpu1 + Rpu2。	读写	0x0
3	保留。	-	-
2	DP_PULLDN_EN : DP 下拉使能	读写	0x0
1	DP_PULLUP_EN : DP 上拉使能	读写	0x0
0	DP_PULLUP_HISEL : 启用第二个 DP 上拉电阻。0 - 拉低 = Rpu2; 1 - 拉低 = Rpu1 + Rpu2	读写	0x0

USB: USBPHY_DIRECT_OVERRIDE 寄存器

偏移: 0x80

描述

usbphy_direct 中各控制项的覆盖使能。

表 416。
USBPHY_DIRECT_OVERRIDE 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	TX_DIFFMODE_OVERRIDE_EN	读写	0x0
14:13	保留。	-	-
12	DM_PULLUP_OVERRIDE_EN	读写	0x0
11	TX_FSSLEW_OVERRIDE_EN	读写	0x0
10	TX_PD_OVERRIDE_EN	读写	0x0
9	RX_PD_OVERRIDE_EN	读写	0x0
8	TX_DM_OVERRIDE_EN	读写	0x0
7	TX_DP_OVERRIDE_EN	读写	0x0
6	TX_DM_OE_OVERRIDE_EN	读写	0x0
5	TX_DP_OE_OVERRIDE_EN	读写	0x0
4	DM_PULLDN_EN_OVERRIDE_EN	读写	0x0
3	DP_PULLDN_EN_OVERRIDE_EN	读写	0x0
2	DP_PULLUP_EN_OVERRIDE_EN	读写	0x0
1	DM_PULLUP_HISEL_OVERRIDE_EN	读写	0x0
0	DP_PULLUP_HISEL_OVERRIDE_EN	读写	0x0

USB: USBPHY_TRIM 寄存器

偏移: 0x84

描述

用于调整 USB 物理层下拉电阻的调节值。

表 417。
USBPHY_TRI
M 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12:8	DM_PULLDN_TRIM : 驱动至 USB PHY 的数值 DM 下拉电阻微调控制 实验数据表明复位值有效，但此寄存器允许根据需要进行调整	读写	0x1f
7:5	保留。	-	-
4:0	DP_PULLDN_TRIM : 驱动至 USB PHY 的数值 DP 下拉电阻微调控制 实验数据表明复位值有效，但此寄存器允许根据需要进行调整	读写	0x1f

USB：INTR 寄存器

偏移：0x8c

描述

原始中断

表 418. INTR
寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	只读	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	只读	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	只读	0x0
16	SETUP_REQ : 设备。来源：SIE_STATUS.SETUP_REC	只读	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	只读	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	只读	0x0
12	BUS_RESET : 来源：SIE_STATUS.BUS_RESET	只读	0x0
11	VBUS_DETECT : 来源：SIE_STATUS.VBUS_DETECTED	只读	0x0
10	STALL : 来源：SIE_STATUS.STALL_REC	只读	0x0
9	ERROR_CRC : 来源：SIE_STATUS.CRC_ERROR	只读	0x0
8	ERROR_BIT_STUFF : 来源：SIE_STATUS.BIT_STUFF_ERROR	只读	0x0
7	ERROR_RX_OVERFLOW : 来源：SIE_STATUS.RX_OVERFLOW	只读	0x0
6	ERROR_RX_TIMEOUT : 来源：SIE_STATUS.RX_TIMEOUT	只读	0x0
5	ERROR_DATA_SEQ : 来源：SIE_STATUS.DATA_SEQ_ERROR	只读	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	只读	0x0

位	描述	类型	复位值
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	只读	0x0
2	HOST_SOF : 主机：每次主机发送 SOF (帧起始) 时触发。 通过读取 SOF_RD 清除	只读	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开 (即 SIE_STATUS.SPEED 变化) 时触发。通过写入 SIE_STATUS.SPEED 清除	只读	0x0

USB: INTE 寄存器

偏移: 0x90

描述

中断使能

表 419. INTE 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	读写	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	读写	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	读写	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	读写	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	读写	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	读写	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	读写	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	读写	0x0
10	STALL : 来源: SIE_STATUSSTALL_REC	读写	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	读写	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	读写	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	读写	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	读写	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	读写	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	读写	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	读写	0x0

位	描述	类型	复位值
2	HOST_SOF : 主机：每次主机发送 SOF（帧起始）时触发。 通过读取 SOF_RD 清除	读写	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	读写	0x0

USB: INTF 寄存器

偏移: 0x94

描述

中断强制

表 420. INTF 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	读写	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	读写	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	读写	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	读写	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	读写	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	读写	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	读写	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	读写	0x0
10	STALL : 来源: SIE_STATUSSTALL_REC	读写	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	读写	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	读写	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	读写	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	读写	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	读写	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	读写	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	读写	0x0
2	HOST_SOF : 主机：每次主机发送 SOF（帧起始）时触发。 通过读取 SOF_RD 清除	读写	0x0

位	描述	类型	复位值
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	读写	0x0

USB: INTS 寄存器

偏移: 0x98

说明

掩码及强制后的中断状态

表 421. INTS 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	只读	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	只读	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	只读	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	只读	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	只读	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	只读	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	只读	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	只读	0x0
10	STALL : 来源: SIE_STATUSSTALL_REC	只读	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	只读	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	只读	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	只读	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	只读	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	只读	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	只读	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	只读	0x0
2	HOST_SOF : 主机：每次主机发送 SOF (帧起始) 时触发。 通过读取 SOF_RD 清除	只读	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	只读	0x0

位	描述	类型	复位值
0	HOST_CONN_DIS: 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	只读	0x0

参考文献

- [1] USB 简明教程
- [2] USB 2.0 规范

4.2. UART

ARM文档

节选自 PrimeCell UART (PL011) 技术参考手册，已获授权使用。

RP2040 拥有两个基于 ARM Primecell UART (PL011) (修订版 r1p5) 的同型号 UART 外设实例。

每个实例支持以下功能：

- 独立的 32×8 发送和 32×12 接收 FIFO 缓冲区
- 可编程波特率发生器，由 `clk_peri` 时钟驱动（详见第 2.15.1 节）
- 发送时添加且接收时去除的标准异步通信位（起始位、停止位、校验位）
- 换行符检测
- 可编程串行接口（5、6、7 或 8 位）
- 1 或 2 个停止位
- 可编程硬件流控制

每个UART可连接至多个GPIO引脚，详见第2.19.2节中的GPIO复用表。

GPIO复用连接前缀为UART实例名 `uart0_` 或 `uart1_`，包含以下内容：

- 发送数据 `tx`（以下章节称为UARTTXD）
- 接收数据 `rx`（以下章节称为UARTRXD）
- 输出流控制 `rts`（以下章节称为nUARTRTS）
- 输入流控制 `cts`（以下章节称为nUARTCTS）。PL011的调制解调器模式及

IrDA模式不支持。

UART`CLK`由`clk_peri`驱动，`PCLK`由系统时钟 `clk_sys`驱动（详见第2.15.1节）。

4.2.1. 概述

UART执行以下操作：

- 对从外围设备接收的数据进行串行转并行转换。
- 对发送至外围设备的数据进行并行转串行转换。

CPU通过AMBA APB接口读写数据及控制/状态信息。发送和接收路径均通过内部FIFO缓存独立缓冲，可分别存储最多32字节的数据，

适用于发送与接收两种模式。

UART具备以下功能：

- 包含可编程波特率发生器，根据UART内部参考时钟输入UARTCLK生成统一的发送和接收内部时钟。
- 功能与行业标准16C650 UART设备相似。
- 在UART模式下支持最高波特率为UARTCLK / 16（125MHz时为7.8 Mbaud）。

UART的操作和波特率由线路控制寄存器UARTLCR_H及波特率除数寄存器（整数波特率寄存器UARTIBRD与小数波特率寄存器UARTFBRD）进行控制。

UART 可产生：

- 来自接收（包括超时）、发送、调制解调器状态及错误条件的可单独屏蔽中断
- 单个合并中断，若任一单个中断被触发，该输出即被置位，且未屏蔽
- 用于与直接存储器访问（DMA）控制器接口的 DMA 请求信号。

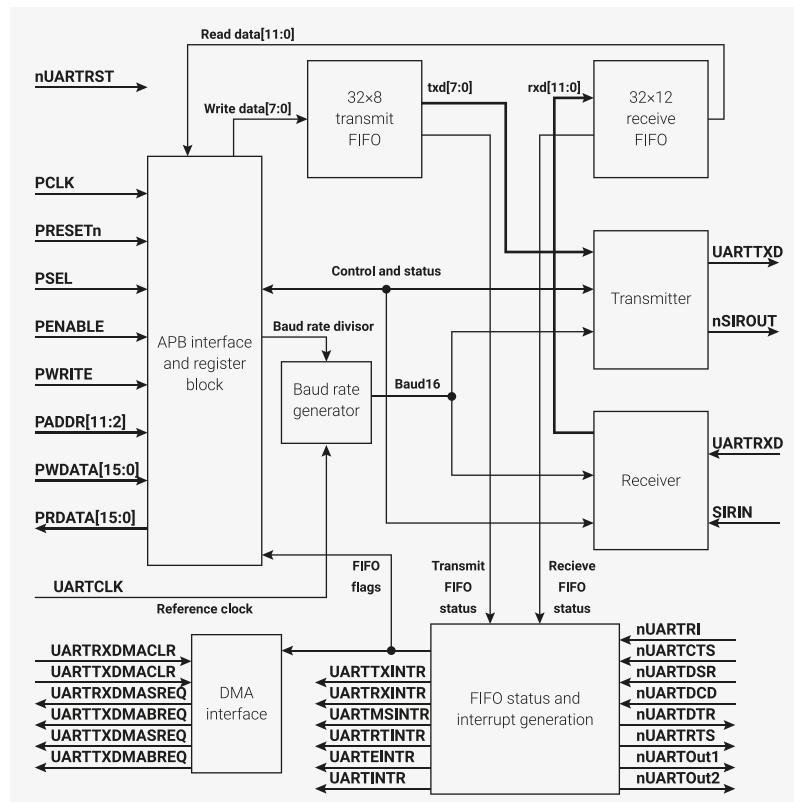
若接收过程中发生帧错误、奇偶校验错误或断续错误，相关错误位会被置位，并存储于 FIFO 中。若发生溢出条件，溢出寄存器位会即时置位，并防止 FIFO 数据被覆盖。

您可以将 FIFO 设置为 1 字节深度，以提供传统的双缓冲 UART 接口。

具备可编程硬件流控制功能，采用 nUARTCTS 输入和 nUARTRTS 输出自动控制串行数据流。

4.2.2. 功能描述

图 59. UART 区块图
。为保持清晰，未显示测试逻辑。



4.2.2.1. AMBA APB 接口

AMBA APB接口产生用于访问状态/控制寄存器及发送和接收FIFO的读写译码。

4.2.2.2. 寄存器块

寄存器块用于存储通过AMBA APB接口写入或将要读取的数据。

4.2.2.3. 波特率发生器

波特率发生器包含自由运行计数器，用于生成内部时钟信号：Baud16和IrLPBaud16。Baud16为UART发送和接收控制提供时序信息。Baud16为一系列脉冲，脉冲宽度为一个UARTCLK时钟周期，频率为波特率的16倍。

4.2.2.4. 发送FIFO

发送FIFO为8位宽、深度为32的位置的FIFO存储缓冲区。通过APB接口写入的CPU数据存储在FIFO中，直至由发送逻辑读取。您可以禁用发送FIFO，使其行为类似于单字节保持寄存器。

4.2.2.5. 接收FIFO

接收FIFO是一个12位宽、深度为32个单元的FIFO存储缓冲区。接收的数据及相应的错误位由接收逻辑存储于接收FIFO中，直至通过APB接口由CPU读取。接收FIFO可被禁用，以作为单字节保持寄存器使用。

4.2.2.6. 发送逻辑

发送逻辑对从发送FIFO读取的数据执行并行转串行转换。控制逻辑依照控制寄存器中设定的配置，输出从起始位开始的串行比特流，数据位最低有效位（LSB）先行，随后为奇偶校验位和停止位。

4.2.2.7. 接收逻辑

接收逻辑在检测到有效的起始脉冲后，对接收的比特流执行串行转并行转换。还会执行溢出、奇偶校验、帧错误检测及线路中断检测，其状态信息随数据一并写入接收FIFO。

4.2.2.8. 中断生成逻辑

UART会产生单独的高电平有效可屏蔽中断。各单独中断请求的逻辑或（OR）功能生成组合中断输出，并连接至处理器中断控制器。

详见第 4.2.6 节。

4.2.2.9. DMA 接口

UART 提供连接至 DMA 控制器的接口，详细描述见第 4.2.5 节的 UART DMA 接口。

4.2.2.10. 同步寄存器与逻辑

UART 支持时钟 PCLK 与 UARTCLK 的异步及同步操作，已实现同步寄存器与握手机制，且始终保持激活。此设计对性能或芯片面积的影响极小。控制信号的同步在数据流双向方向进行，即从 PCLK 域到 UARTCLK 域，以及从 UARTCLK 域到 PCLK 域。

4.2.3. 操作

4.2.3.1. 时钟信号

为UARTCLK选择的频率必须满足所需的波特率范围：

- FUARTCLK（最小值） $\geq 16 \times \text{baud_rate}$ （最大值）
- FUARTCLK（最大值） $\leq 16 \times 65535 \times \text{baud_rate}$ （最小值）

例如，对于110波特到460800波特的波特率范围，UARTCLK频率必须在7.3728MHz至115.34MHz之间。

UARTCLK的频率还必须在所有所用波特率的规定误差范围内。

对于PCLK与UARTCLK的时钟频率比也有约束。UARTCLK频率不得超过PCLK频率的5/3倍：

- FUARTCLK $\leq \frac{5}{3} \times \text{FPCLK}$

例如，在UART模式下，若UARTCLK为14.7456MHz，要生成921600波特，PCLK必须大于或等于8.85276MHz，以确保UART有足够时间将接收数据写入接收FIFO。

4.2.3.2. UART 操作

控制数据写入 UART 线路控制寄存器 UARTLCR。该寄存器内部宽度为 30 位，但通过 APB 接口对以下寄存器进行写访问：

UARTLCR_H 寄存器定义：

- 传输参数
- 字长
- 缓冲模式
- 发送停止位数
- 奇偶校验模式
- 断线生成。

UARTIBRD 寄存器定义整数波特率分频器，UARTFBRD 寄存器定义小数波特率分频器。

4.2.3.2.1. 小数波特率分频器

波特率分频值为一个 22 位数字，由 16 位整数部分和 6 位小数部分组成。该值用于波特率发生器确定比特周期。小数波特率分频器允许使用任何频率大于 3.6864MHz 的时钟作为 UARTCLK，同时仍能生成所有标准波特率。

16 位整数写入整数波特率寄存器 UARTIBRD。6 位小数部分写入小数波特率寄存器 UARTFBRD。波特率除数与 UARTCLK 之间的关系如下：

波特率除数 = $\text{UARTCLK}/(16 \times \text{波特率}) = BRD_I + BRD_F$ 其中 BRD_I 为整数部分， BRD_F 为如图 60 所示由小数点分隔的小数部分。

图 60. 波特率除数。



可通过取所需波特率除数的小数部分并乘以 64（即，其中为 UARTFBRD 寄存器的位宽），再加 0.5 以进行四舍五入，计算该 6 位数 2^n ： $n = \text{integer}(BRD_F \times 64 + 0.5)$

内部生成一个时钟使能信号 Baud16，该信号为一串宽度为一个 UARTCLK 的脉冲，其平均频率为所需波特率的 16 倍。该信号随后被除以 16，作为发送时钟。波特率除数较低时，位周期较短；波特率除数较高时，位周期较长。

4.2.3.2.2 数据传输或接收

接收或发送的数据存储在两个32字节的FIFO中，但接收FIFO每个字符额外包含四位状态信息。对于发送，数据写入发送FIFO。若UART已启用，则会根据行控制寄存器UARTLCR_H中指定的参数启动数据帧的发送。数据持续传输，直至发送FIFO无数据。只要数据写入发送FIFO（即FIFO非空），BUSY信号即被置高，并在数据传输过程中保持高电平。仅当发送FIFO为空，且最后一个字符（包括停止位）已从移位寄存器发送完毕时，BUSY信号才会被置低。即使UART可能已不再启用，BUSY信号仍可能保持高电平。

对于每个数据样本，采集三次读数，并保留多数值。以下段落中定义了中间采样点，并在其两侧各取一个样本。

当接收器处于空闲状态（UARTRXD持续为1，处于标记状态）且在数据输入端检测到低电平（表示接收到起始位）时，由Baud16使能时钟的接收计数器启动计数，数据在UART模式下于计数器的第八个周期采样，或在SIR模式下于计数器的第四个周期采样，以适应较短的逻辑0脉冲（位周期中点）。

若Baud16的第八个周期时UARTRXD仍为低电平，则判定起始位有效，否则视为误起始位并予以忽略。

若起始位有效，依据编程设置的数据字符长度，随后每隔16个Baud16周期采样一次数据位（即一个位周期后采样）。如果启用了奇偶校验模式，则随后检查奇偶校验位。

最后，如果UARTRXD为高电平，则确认有效的停止位，否则发生帧错误。当接收到完整字时，数据存储于接收FIFO中，且与该字相关的任何错误位也一并存储。

4.2.3.2.3. 错误位

三个错误位存储于接收FIFO的第10至第8位，并关联至特定字符。另有一项溢出错误，存储于接收FIFO的第11位。

4.2.3.2.4. 溢出位

溢出位不与接收FIFO中的字符关联。当FIFO已满且移位寄存器中完整接收下一字符时，溢出错误位被置位。移位寄存器中的数据将被覆盖，但不会写入FIFO。当接收FIFO有空位且接收到另一字符时，溢出位状态会随所接收字符一并复制入接收FIFO。随后清除溢出状态。表422列出了接收FIFO的位功能。

表422。接收
FIFO位功能

FIFO位	功能
11	溢出指示符
10	断帧错误
9	奇偶校验错误
8	帧错误
7:0	接收数据

4.2.3.2.5. 禁用FIFO

此外，您可以禁用FIFO。在此情况下，UART的发送和接收端各配备1字节保持寄存器（即FIFO的最低条目）。当接收到一个字且前一个字尚未读取时，溢出位将被置位。在本实现中，FIFO未被物理禁用，而是通过操作标志位以模拟1字节寄存器的行为。FIFO禁用时，对数据寄存器的写操作会绕过保持寄存器，除非发送移位寄存器已在使用。

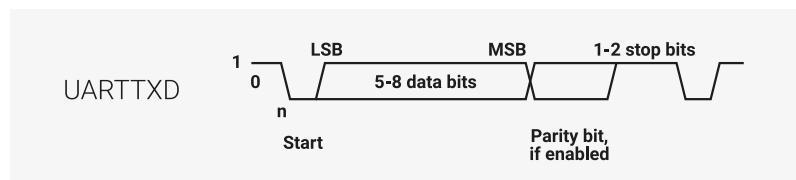
4.2.3.2.6. 系统及诊断环回测试

您可以通过将控制寄存器 UARTCR 中的循环回环使能（LBE）位设置为 1 来执行 UART 数据的回环测试，[UARTCR](#)。

在 UARTRXD 上发送的数据将在 UARTRXD 输入端接收。

4.2.3.3 UART 字符帧

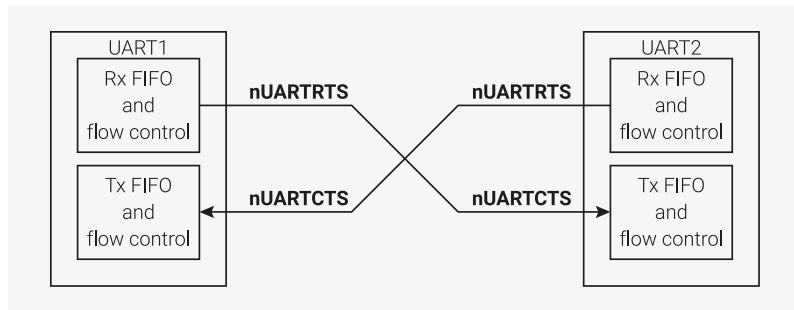
图 61 UART
字符帧。



4.2.4. UART 硬件流控制

硬件流控功能完全可选，允许您通过 nUARTRTS 输出和 nUARTCTS 输入信号控制串行数据流。图 62 展示了两个设备如何利用硬件流控进行通信。

图 62。两个相似设备之间的硬件流控制。



启用 RTS 流控后，nUARTRTS 信号将保持有效，直到接收 FIFO 达到预设水位线。启用 CTS 流控后，只有在 nUARTCTS 信号有效时，发送端才能传输数据。

硬件流控可通过控制寄存器 UARTCR 中的 RTSEn 和 CTSEn 位选择。表 423 列出了同时启用或分别启用 RTS 和 CTS 流控时各个位的设置方法。

表 423。用于启用和禁用硬件流控制的控制位。

UARTCR 寄存器位		
CTSEn	RTSEn	描述
1	1	同时启用 RTS 和 CTS 流控制
1	0	仅启用 CTS 流控制
0	1	仅启用 RTS 流控制
0	0	同时禁用 RTS 和 CTS 流控制

① 注意

启用 RTS 流控制时，软件无法使用控制寄存器 UARTCR 中的 RTSEn 位来控制 nUARTRTS 的状态。

4.2.4.1 RTS 流控制

RTS 流控制逻辑与可编程接收 FIFO 的水位线级别相关联。启用 RTS 流控制时，nUARTRTS 信号被激活，直到接收 FIFO 填满至水位线水平。当接收 FIFO 达到水位线水平时，nUARTRTS 信号被取消激活，表示接收缓冲区已无更多数据接收空间。预计当前字符传输完成后，数据传输将停止。

当接收 FIFO 中的数据被读取，使其填充量降至低于水位线时，nUARTRTS 信号将重新断言。如果 RTS 流控被禁用且 UART 仍处于使能状态，则数据将继续接收，直到接收 FIFO 已满或没有更多数据传入。

4.2.4.2. CTS流控

如果启用 CTS 流控，则发送器在发送下一个字节前会检测 nUARTCTS 信号。若 nUARTCTS 信号被断言，则发送该字节；否则不进行传输。

只要 nUARTCTS 被断言且发送 FIFO 非空，数据将持续传输。如果发送 FIFO 为空且 nUARTCTS 信号被断言，则不发送数据。

如果 nUARTCTS 信号被取消断言且启用 CTS 流控，则当前字符传输完成后停止传输。如果 CTS 流控被禁用且 UART 被启用，则数据继续传输直至发送 FIFO 为空。

4.2.5. UART DMA 接口

UART 提供一个接口，用于连接 DMA 控制器。UART 的 DMA 操作通过 DMA 控制寄存器 UARTDMACR 进行控制。DMA 接口包含以下信号：

接收：

UARTRXDMASREQ

单字符 DMA 传输请求，由 UART 断言。对于接收，单字符最多包含 12 位。
当接收 FIFO 至少含有一个字符时，该信号被断言。

UARTRXDMABREQ

突发 DMA 传输请求，由 UART 断言。当接收 FIFO 中的字符数超过编程设定的水位线时，该信号被断言。您可以通过中断 FIFO 水平选择寄存器 UARTIFLS 为每个 FIFO 编程水位线。

UARTRXDMACLR

DMA 请求清除信号，由 DMA 控制器断言，用以清除接收请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

发送：

UARTTXDMASREQ

单字符 DMA 传输请求，由 UART 发出。传输时，一个字符由最多八位组成。当发送 FIFO 中至少存在一个空位时，该信号被置位。

UARTTXDMABREQ

突发 DMA 传输请求，由 UART 发出。当发送 FIFO 中的字符数少于水位线时，该信号被置位。您可以通过中断 FIFO 级别选择寄存器 UARTIFLS 为每个 FIFO 设置水位线级别。

UARTTXDMACLR

DMA 请求清除信号，由 DMA 控制器发出，用以清除发送请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

突发传输请求信号与单次传输请求信号并非互斥，可同时置位。例如，当接收 FIFO 中的数据量多于水位线时，突发传输请求与单次传输请求均被置位。当接收 FIFO 中剩余数据量少于水位线时，仅单次传输请求被置位。此功能适用于串流中剩余待接收字符数少于突发传输量的情况。

例如，若需接收 19 个字符且水印级别设置为四。随后，DMA 控制器执行四次四字符突发传输及三次单字符传输以完成数据流。

i 注意

对于剩余的三个字符，UART 无法发出突发请求信号。

每个请求信号保持有效状态，直至相应的 DMA CLR 信号被置位。在请求清除信号取消置位后，请求信号可根据前述条件重新激活。若 UART 被禁用，或 DMA 控制寄存器 UARTDMACR 中的相关 DMA 使能位 TXDMAE 或 RXDMAE 被清除，所有请求信号均被取消置位。

若禁用 UART 中的 FIFO，则其工作于字符模式，仅支持 DMA 单字符传输模式，因每次仅能传输一个字符至或从 FIFO。UARTRXDMASREQ 与 UARTTXDMASREQ 为唯一可置位的请求信号。有关禁用 FIFO 的信息，请参见线路控制寄存器 UARTLCR_LH。

当 UART 处于启用 FIFO 模式时，数据传输可根据编程的水位线水平及 FIFO 中的数据量，通过单次或突发传输完成。表 4-24 列出了根据水位线水平，针对发送和接收 FIFO 的 UARTRXDMABREQ 和 UARTTXDMABREQ 触发点。

表 424。发送和接收 FIFO 的 DMA 触发点。

水位线	突发长度	
	发送（空闲位置数）	接收（填充位置数量）
1/8	28	4
1/4	24	8
1/2	16	16
3/4	8	24
7/8	4	28

此外，DMA 控制寄存器 UARTRDMACR 中的 DMAONERR 位支持接收错误中断 UARTEINTR 的使用。当 UART 错误中断 UARTEINTR 被触发时，该位可屏蔽 DMA 接收请求输出 UARTRXDMASREQ 或 UARTRXDMABREQ。DMA 接收请求输出保持非激活状态，直至清除 UARTEINTR。DMA 发送请求输出不受影响。

图63. DMA 传输波形。

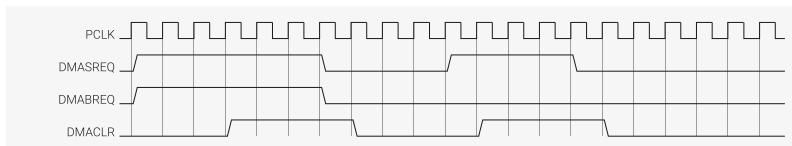


图63显示了单次传输请求及携带相应DMACLR信号的突发传输请求的时序图。所有信号均与PCLK同步。为简明起见，假设DMA控制器内的请求信号未进行同步处理。

4.2.6. 中断

UART产生十一种可屏蔽中断。在RP2040上，仅连接了组合中断输出UARTINTR。

您可通过修改中断屏蔽设置/清除寄存器UARTIMSC中的屏蔽位来启用或禁用各个中断。相应屏蔽位置高即启用该中断。

单独中断输出及组合中断输出的提供，使您能够使用全局中断服务程序或模块化设备驱动程序来处理中断。

发送和接收数据流中断UARTRXINTR与UARTRXINTR已与状态中断分离。此功能使您能够使用 UARTRXINTR 和 UARTRXINTR，以便根据 FIFO 触发电平读取或写入数据。

当接收数据发生错误时，错误中断 UARTEINTR 会被触发。可能存在多种错误情况。

调制解调器状态中断 UARTMSINTR 是由所有独立调制解调器状态信号组合而成的中断。

各个中断源的状态可以从原始中断状态寄存器 UARTRIS 中读取，
也可以从屏蔽中断状态寄存器 UARTMIS 中读取。

4.2.6.1. UARTMSINTR

当任何调制解调器状态信号（nUARTCTS、nUARTDCD、nUARTDSR 和 nUARTRI）发生变化时，调制解调器状态中断即被置位。根据引发中断的调制解调器状态信号，通过向中断清除寄存器 UARTRICR 中相应的位写入 1 来清除该中断。

4.2.6.2. UARTRXINTR

当发生以下任一事件时，接收中断状态将改变：

- 如果启用了FIFO且接收FIFO达到预设的触发级别。发生此情况时，接收中断被置为高电平。接收中断通过从接收FIFO读取数据直到其低于触发级别，或通过清除中断予以清除。
- 如果禁用FIFO（深度为一个位置），且接收到数据使该位置被填满，则接收中断被置为高电平。接收中断通过对接收FIFO执行单次读取，或通过清除中断予以清除。

4.2.6.3. UARTTXINTR

当发生以下任一事件时，发送中断状态将改变：

- 如果启用了FIFO且发送FIFO低于或等于预设的触发级别，则发送中断被置为高电平。发送中断通过向发送FIFO写入数据直到其超过触发级别，或通过清除中断予以清除。
- 如果FIFO被禁用（深度为一个位置）且发射器的单个位置中无数据，则发送中断被置为高电平。该中断可通过对发送FIFO执行单次写操作或清除中断来清除。

要更新发送FIFO，必须：

- 向发送FIFO写入数据，可以在启用UART和中断之前，或启用UART和中断之后执行。

i 注意

发送中断基于电平的跳变触发，而非电平本身。当中断与UART在向发送FIFO写入任何数据之前被启用时，中断不会被置位。只有在写入数据离开发送FIFO的单个位置且该位置变为空时，中断才会被置位。

4.2.6.4. UARTRTINTR

当接收FIFO非空且在32位周期内未接收到更多数据时，接收超时中断被触发。接收超时中断将在FIFO通过读取所有数据（或读取保持寄存器）变为空，或写1至中断清除寄存器UARTICR的对应位时被清除。

4.2.6.5. UARTEINTR

当UART接收数据发生错误时，会触发错误中断。该中断可能由多种不同的错误情况引起：

- 帧错误
- 奇偶校验错误
- 断帧
- 溢出错误。

您可以通过读取原始中断状态寄存器（Raw Interrupt Status Register，UARTRIS）或屏蔽中断状态寄存器（Masked Interrupt Status Register，UARTMIS）来确定中断原因。中断可通过向中断清除寄存器（Interrupt Clear Register，UARTICR）相关位写入数据来清除（第7至10位为错误清除位）。

4.2.6.6. UARTINTR

这些中断也会被合并为一个输出，该输出为各个屏蔽中断源的逻辑或。您可以将此输出连接至系统中断控制器，以针对单个外设实现另一级别的屏蔽。

只要任一单独中断被触发且启用，合并的UART中断即被触发。

4.2.7. 程序员模型

SDK提供了一个 `uart_init` 函数，用于配置特定波特率的UART。UART初始化完成后，用户必须将GPIO引脚配置为 `UART_T` 与 `UART_RX`。有关GPIO功能选择的更多信息，请参见第2.19.5.1节。

初始化UART时，`uart_init` 函数按以下步骤执行：

- 取消复位信号
- 启用 `clk_peri`
- 在控制寄存器中设置使能位
- 启用FIFO
- 设置波特率分频器
- 配置格式

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c 第42至92行

```

42 uint uart_init(uart_inst_t *uart, uint baudrate) {
43     invalid_params_if(HARDWARE_UART, uart != uart0 && uart != uart1);
44
45     if (uart_clock_get_hz(uart) == 0) {
46         return 0;
47     }
48
49     uart_reset(uart);
50     uart_unreset(uart);
51
52     uart_set_translate_crlf(uart, PICO_UART_DEFAULT_CRLF);
53
54     // 所有LCR寄存器写入操作必须在启用UART之前完成
55     uint baud = uart_set_baudrate(uart, baudrate);
56
57     // 内联uart_set_format()调用，因为我们不需要对CR进行禁用/重新启用
58     // 保护，且许多人可能永远不会再调用它，
59     // 因此通用函数没有实用价值，且其内联代码远小于函数本身，
60     // 仅包含少量指令。
61
62     // UART_UARTLCR_H_FEN_BITS设置合并处理，因为它们位于同一寄存器
63 #ifdef 0
64     uart_set_format(uart, 8, 1, UART_PARITY_NONE);
65     // 启用FIFO（必须在设置UARTEN之前完成，因为这是一次LCR寄存器访问）
66     hw_set_bits(&uart_get_hw(uart)->lcr_h, UART_UARTLCR_H_FEN_BITS);
67 #else
68     uint data_bits = 8;
69     uint stop_bits = 1;
70     uint parity = UART_PARITY_NONE;
71     hw_write_masked(&uart_get_hw(uart)->lcr_h,
72                     ((data_bits - 5u) << UART_UARTLCR_H_WLEN_LSB) |
73                     ((stop_bits - 1u) << UART_UARTLCR_H_STP2_LSB) |
74                     (bool_to_bit(parity != UART_PARITY_NONE) << UART_UARTLCR_H_PEN_LSB) |
75                     (bool_to_bit(parity == UART_PARITY_EVEN) << UART_UARTLCR_H_EPS_LSB) |

```

```

76         UART_UARTLCR_H_FEN_BITS,
77         UART_UARTLCR_H_WLEN_BITS | UART_UARTLCR_H_STP2_BITS |
78         UART_UARTLCR_H_PEN_BITS | UART_UARTLCR_H_EPS_BITS |
79         UART_UARTLCR_H_FEN_BITS);
80 #endif
81
82 // 启用 UART，包括发送 (TX) 和接收 (RX)
83 uart_get_hw(uart)->cr = UART_UARTCR_UARTEN_BITS | UART_UARTCR_TXE_BITS |
84     UART_UARTCR_RXE_BITS;
85 // 始终启用 DREQ 信号——若 DMA 未监听，启用无害
86 uart_get_hw(uart)->dmacr = UART_UARTDMACR_TXDMAE_BITS | UART_UARTDMACR_RXDMAE_BITS;
87
88 return baud;
89 }
```

4.2.7.1. 波特率计算

UART 波特率是通过除以 `clk_peri` 计算得出。

若所需波特率为 115200 且 UARTCLK 为 125MHz，则：

$$\text{波特率除数} = (125 * 10^6) / (16 * 115200) \approx 67.817$$

因此，BRDI = 67，BRDF = 0.817，

因此，小数部分 m = integer((0.817 * 64) + 0.5) = 52

生成的波特率分频值 = $67 + 52/64 = 67.8125$

$$\text{生成的波特率} = (125 * 10^6) / (16 * 67.8125) \approx 115207$$

$$\text{误差} = (\text{abs}(115200 - 115207) / 115200) * 100 \approx 0.006\%$$

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c，第155至180行

```

155 uint uart_set_baudrate(uart_inst_t *uart, uint baudrate) {
156     invalid_params_if(HARDWARE_UART, baudrate == 0);
157     uint32_t baud_rate_div = (8 * uart_clock_get_hz(uart) / baudrate) + 1;
158     uint32_t baud_ibrd = baud_rate_div >> 7;
159     uint32_t baud_fbrd;
160
161     if (baud_ibrd == 0) {
162         baud_ibrd = 1;
163         baud_fbrd = 0;
164     } else if (baud_ibrd >= 65535) {
165         baud_ibrd = 65535;
166         baud_fbrd = 0;
167     } else {
168         baud_fbrd = (baud_rate_div & 0x7f) >> 1;
169     }
170
171     uart_get_hw(uart)->ibrd = baud_ibrd;
172     uart_get_hw(uart)->fbrd = baud_fbrd;
173
174 // PL011 需要一个 (伪) LCR_H 写入以锁存分频值。
175 // 此处不应实际更改 LCR_H 的内容。
176     uart_write_lcr_bits_masked(uart, 0, 0);
177
178 // 参见数据手册
179     return (4 * uart_clock_get_hz(uart)) / (64 * baud_ibrd + baud_fbrd);
180 }
```

4.2.8. 寄存器列表

UART0 和 UART1 寄存器的基地址分别为 `0x40034000` 和 `0x40038000` (在 SDK 中定义为 `UART0_BASE` 和 `UART1_BASE`)

◦

表 425. UART 寄存器列表

偏移量	名称	说明
0x000	<code>UARTDR</code>	数据寄存器, <code>UARTDR</code>
0x004	<code>UARTRSR</code>	接收状态寄存器／错误清除寄存器 <code>UARTRSR/UARTECR</code>
0x018	<code>UARTFR</code>	标志寄存器, <code>UARTFR</code>
0x020	<code>UARTILPR</code>	红外低功耗计数器寄存器, <code>UARTILPR</code>
0x024	<code>UARTIBRD</code>	整数波特率寄存器, <code>UARTIBRD</code>
0x028	<code>UARTFBRD</code>	分数波特率寄存器, <code>UARTFBRD</code>
0x02c	<code>UARTLCR_H</code>	线路控制寄存器, <code>UARTLCR_H</code>
0x030	<code>UARTCR</code>	控制寄存器, <code>UARTCR</code>
0x034	<code>UARTIFLS</code>	中断 FIFO 水平选择寄存器, <code>UARTIFLS</code>
0x038	<code>UARTIMSC</code>	中断屏蔽设置/清除寄存器, <code>UARTIMSC</code>
0x03c	<code>UARTRIS</code>	原始中断状态寄存器, <code>UARTRIS</code>
0x040	<code>UARTMIS</code>	掩码中断状态寄存器, <code>UARTMIS</code>
0x044	<code>UARTICR</code>	中断清除寄存器, <code>UARTICR</code>
0x048	<code>UARTDMACR</code>	DMA 控制寄存器, <code>UARTDMACR</code>
0xfe0	<code>UARTPERIPHID0</code>	UARTPeriphID0 寄存器
0xfe4	<code>UARTPERIPHID1</code>	UARTPeriphID1 寄存器
0xfe8	<code>UARTPERIPHID2</code>	UARTPeriphID2 寄存器
0xfc	<code>UARTPERIPHID3</code>	UARTPeriphID3 寄存器
0xff0	<code>UARTPCELLID0</code>	UARTPCellID0 寄存器
0xff4	<code>UARTPCELLID1</code>	UARTPCellID1 寄存器
0xff8	<code>UARTPCELLID2</code>	UARTPCellID2 寄存器
0ffc	<code>UARTPCELLID3</code>	UARTPCellID3 寄存器

UART：UARTDR 寄存器

偏移: 0x000

描述

数据寄存器, `UARTDR`

表 426. `UARTDR` 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	OE: 溢出错误。当接收到数据且接收 FIFO 已满时, 该位置为 1。一旦 FIFO 有空位且可写入新字符, 该位清零为 0。	只读	-

位	描述	类型	复位值
10	BE : 断帧错误。当检测到断帧条件，即接收数据输入线长时间保持低电平，超过完整字传输时间（包括起始位、数据位、校验位和停止位）时，该位置为1。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。断帧发生时，仅加载一个0字符入 FIFO。仅当接收数据输入恢复为高电平（标记状态），且接收到下一个有效起始位后，下一字符方被允许进入。	只读	-
9	PE : 校验错误。当该位为1时，表示接收的字符奇偶校验与 UARTECR_H（线路控制寄存器）中 EPS 和 SPS 位规定的不符。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。	只读	-
8	FE : 帧错误。当此位设置为1时，表示接收的字符没有有效停止位（有效停止位为1）。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关联。	只读	-
7:0	DATA : 接收（读取）数据字符。发送（写入）数据字符。	RWF	-

UART: UARTRSR 寄存器

偏移: 0x004

描述

接收状态寄存器/错误清除寄存器，UARTRSR/UARTECR

表427. UARTRSR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	OE : 溢出错误。若接收到数据且 FIFO 已满，该位被置为1。通过写入 UARTECR 可将该位清零。FIFO 内容保持有效，因为 FIFO 已满时不再写入新数据，仅覆盖移位寄存器的内容。CPU 必须读取数据以清空 FIFO。	WC	0x0
2	BE : 断帧错误。若检测到中断条件，该位被置为1，表示接收数据输入被保持为低电平，持续时间超过一个完整字的传输时间（定义为起始位、数据位、奇偶校验位和停止位）。该位在写入 UARTECR 后清零。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。发生 break 时，FIFO 中仅加载一个0字符。只有当接收数据输入变为1（标记状态）且接收到下一个有效起始位后，下一字符才被使能。	WC	0x0
1	PE : 校验错误。该位置1表示接收数据字符的奇偶校验与 UARTECR_H（线路控制寄存器）中的 EPS 和 SPS 位设置不符。通过写入 UARTECR 可将该位清零。 在 FIFO 模式下，该错误与 FIFO 顶部的字符相关联。	WC	0x0
0	FE : 帧错误。当此位设置为1时，表示接收的字符没有有效停止位（有效停止位为1）。该位通过写入 UARTECR 清零。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。	WC	0x0

UART: UARTFR 寄存器

偏移: 0x018

描述

标志寄存器，UARTFR

表428. UARTFR寄存器

位	描述	类型	复位值
31:9	保留。	-	-
8	RI : 环指示符。该位为 UART 环指示器 nUARTRI (调制解调器状态输入) 的反码。即当 nUARTRI 为 LOW 时, 该位为 1。	只读	-
7	TXFE : 发送 FIFO 为空。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当发送保持寄存器为空时该位被置位。若 FIFO 启用, 则当发送 FIFO 为空时 TXFE 位被置位。该位不指示发送移位寄存器中是否存在数据。	只读	0x1
6	RXFF : 接收 FIFO 已满。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当接收保持寄存器已满时该位被置位。若 FIFO 启用, 则当接收 FIFO 已满时 RXFF 位被置位。	只读	0x0
5	TXFF : 发送 FIFO 已满。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当发送保持寄存器已满时该位被置位。如果启用了 FIFO, 当传输FIFO满时, TXFF位将被置位。	只读	0x0
4	RXFE : 接收FIFO为空。该位的含义取决于UARTLCR_H寄存器中FEN位的状态。若FIFO被禁用, 当接收保持寄存器为空时, 该位将被置位。如果启用了FIFO, 当接收FIFO为空时, RXFE位将被置位。	只读	0x1
3	BUSY : UART 忙碌。当该位被置为 1 时, 表示 UART 正在忙于传输数据。该位将保持置位状态, 直至包含所有停止位的完整字节从移位寄存器发送完成。不论 UART 是否启用, 只要传输 FIFO 变为非空, 该位即被置位。	只读	0x0
2	DCD : 数据信号载波检测。该位为 UART 数据载波检测输入信号 nUARTDCD 的反码, 调制解调器状态输入。即当 nUARTDCD 为低电平时该位为 1。	只读	-
1	DSR : 数据集准备就绪。该位为 UART 数据集准备就绪信号 nUARTDSR 的反码, 调制解调器状态输入。即当 nUARTDSR 为低电平时, 该位为 1。	只读	-
0	CTS : 清除发送。该位为 UART 清除发送信号 nUARTCTS 的反码, 调制解调器状态输入。即当 nUARTCTS 为低电平时, 该位为 1。	只读	-

UART：UARTILPR寄存器

偏移: 0x020

描述

红外低功耗计数器寄存器，UARTILPR

表429. UARTILPR寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	ILPDVSR : 8位低功耗除数值。这些位在复位时全部清零。	读写	0x00

UART：UARTIBRD寄存器

偏移: 0x024

描述

整数波特率寄存器，UARTIBRD

表430. UARTIBRD寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	BAUD_DIVINT : 整数波特率除数。这些位在复位时全部清零。	读写	0x0000

UART：UARTFBRD寄存器

偏移：0x028

描述

分数波特率寄存器，UARTFBRD

表431. UARTFBRD寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	BAUD_DIVFRAC : 分数波特率除数。这些位在复位时全部清零。	读写	0x00

UART：UARTLCR_H 寄存器

偏移：0x02c

描述

线路控制寄存器，UARTLCR_H

表432。UARTLCR_H 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	SPS : 停用奇偶校验选择。0 = 停用奇偶校验；1 = 以下任一情况：* 当 EPS 位为0时，奇偶校验位以1传输和检测；* 当 EPS 位为1时，奇偶校验位以0传输和检测。当 PEN 位禁用奇偶校验的生成和检测时，此位无效。	读写	0x0
6:5	WLEN : 字长。这些位指示帧中传输或接收的数据位数，如下：b11 = 8位，b10 = 7位，b01 = 6位，b00 = 5位。	读写	0x0
4	FEN : 启用FIFO: 0 = 禁用FIFO（字符模式），即FIFO变为1字节深的保持寄存器；1 = 启用发送和接收FIFO缓冲区（FIFO模式）。	读写	0x0
3	STP2 : 选择两个停止位。如果此位设置为1，将在帧末尾发送两个停止位。接收逻辑不会检测是否接收了两个停止位。	读写	0x0
2	EPS : 奇偶校验选择。控制UART在发送和接收过程中使用的奇偶校验类型：0 = 奇校验。UART会生成或检测数据及校验位中的奇数个1。1 = 偶校验。UART会生成或检测数据及校验位中的偶数个1。当 PEN 位禁用奇偶校验的生成和检测时，此位无效。	读写	0x0
1	PEN : 奇偶校验使能；0 = 禁用奇偶校验且不在数据帧中添加校验位，1 = 启用奇偶校验的生成和检测。	读写	0x0

位	描述	类型	复位值
0	BRK : 发送中断信号。如果此位设置为1，则在完成当前字符的传输后，UART TTXD输出端持续输出低电平。为正确执行中断命令，软件必须将此位至少保持两个完整帧的时间。在正常使用情况下，该位必须清零（置为0）。	读写	0x0

UART: UARTCR 寄存器

偏移: 0x030

描述

控制寄存器，UARTCR

表 433. UARTCR 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	CTSEN : CTS 硬件流控制使能。当该位被置为1时，启用CTS硬件流控制。仅当nUARTCTS信号被断言时，才发送数据。	读写	0x0
14	RTSEN : RTS 硬件流控制使能。当该位被置为1时，启用RTS硬件流控制。仅当接收FIFO有空间时，才请求数据接收。	读写	0x0
13	OUT2 : 该位为UART Out2 (nUARTOut2) 调制解调器状态输出的反码。即当该位被设置为1时，输出值为0。对于DTE，该位可用作环指示 (RI)。	读写	0x0
12	OUT1 : 该位为UART Out1 (nUARTOut1) 调制解调器状态输出的反码。亦即，当该位被置为1时，输出为0。对于DTE，该位可用作数据载波检测 (DCD)。	读写	0x0
11	RTS : 请求发送。该位为UART请求发送信号nUARTRTS (调制解调器状态输出) 之反码。亦即，当该位被置为1时，nUARTRTS为低电平。	读写	0x0
10	DTR : 数据传输准备。该位为UART数据传输准备信号nUARTDTR (调制解调器状态输出) 之反码。亦即，当该位被置为1时，nUARTDTR为低电平。	读写	0x0
9	RXE : 接收使能。当该位被置为1时，UART接收部分被启用。数据接收可基于SIREN位设置，接收UART信号或SIR信号。当UART在接收过程中被禁用时，会完成当前字符的接收后再停止。	读写	0x1
8	TXE : 发送使能。若此位被置为1，则UART的发送部分被使能。数据传输根据SIREN位的设置，通过UART信号或SIR信号进行。若UART在传输过程中被禁用，则会完成当前字符传输后停止。	读写	0x1

位	描述	类型	复位值
7	LBE : 回环使能。若此位设置为1，且SIREN位为1，同时测试控制寄存器UARTTCR中的SIRTEST位为1，则nSIROUT路径将被反相，并反馈至SIRIN路径。测试寄存器中的SIRTEST位必须设置为1，方可覆盖正常的半双工SIR操作。此为正常操作期间访问测试寄存器的必要条件，回环测试结束后，须将SIRTEST清零。该功能减少了系统测试期间所需的外部耦合量。如果该位被设置为1，且SIRTEST位被设置为0，则UARTTXD路径会传递至UARTRXD路径。无论在SIR模式还是UART模式，当该位被设置时，调制解调器输出同样会传递至调制解调器输入。该位在复位时被清零（设置为0），以禁用环回功能。	读写	0x0
6:3	保留。	-	-
2	SIRLP : 低功耗IrDA SIR模式。该位用于选择IrDA编码模式。如果该位被清零（设置为0），则低电平比特以宽度为比特周期3/16的高电平脉冲传输。如果该位被设置为1，低电平比特的脉冲宽度为IrLPBaud16输入信号周期的3倍，与所选比特率无关。设置该位可降低功耗，但可能会缩短传输距离。	读写	0x0
1	SIREN : SIR使能：0 = 禁用IrDA SIR ENDEC。nSIROUT保持低电平（未产生光脉冲），SIRIN上的信号跳变无效。1 = 启用IrDA SIR 编解码器。数据通过nSIROUT和SIRIN进行传输和接收。UARTTXD保持高电平，处于标记状态。UARTRXD或调制解调器状态输入端的信号跳变无效。如果UARTEN位禁用UART，则此位无效。	读写	0x0
0	UARTEN : UART使能，0 = UART已禁用。如果在传输或接收过程中禁用UART，则会完成当前字符传输后停止。1 = UART已启用。数据传输和接收根据SIREN位设置，采用UART信号或SIR信号。	读写	0x0

UART: UARTIFLS 寄存器

偏移: 0x034

描述

中断 FIFO 水平选择寄存器，UARTIFLS

表 434. UARTIFLS 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:3	RXIFLSEL : 接收中断 FIFO 水平选择。接收中断触发点如下：b000 = 接收 FIFO 达到或超过 1/8 满；b001 = 接收 FIFO 达到或超过 1/4 满；b010 = 接收 FIFO 达到或超过 1/2 满。 b011 = 接收 FIFO 满 >= 3/4 b100 = 接收 FIFO 满 >= 7/8 b101-b111 = 保留。	读写	0x2
2:0	TXIFLSEL : 发送中断 FIFO 水平选择。发送中断的触发点如下：b000 = 发送 FIFO 满 <= 1/8 b001 = 发送 FIFO 满 <= 1/4 b010 = 发送 FIFO 满 <= 1/2 b011 = 发送 FIFO 满 <= 3/4 b100 = 发送 FIFO 满 <= 7/8 b101-b111 = 保留。	读写	0x2

UART: UARTIMSC 寄存器

偏移: 0x038

说明

中断屏蔽设置/清除寄存器, UARTIMSC

表 435. UARTIMSC
寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEIM: 溢出错误中断屏蔽。读取操作返回当前 UARTOEINTR 中断的屏蔽状态。写入 1 时, 设置 UARTOEINTR 中断屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
9	BEIM: 中断错误中断掩码。读取返回UARTBEINTR中断的当前掩码。写入1时, UARTBEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
8	PEIM: 奇偶校验错误中断掩码。读取返回UARTPEINTR中断的当前掩码。写入1时, UARTPEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
7	FEIM: 帧错误中断掩码。读取返回UARTFEINTR中断的当前掩码。写入1时, UARTFEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
6	RTIM: 接收超时中断掩码。读取返回UARTRTINTR中断的当前掩码。写入1时, UARTRTINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
5	TXIM: 发送中断屏蔽。读取操作返回 UARTRXINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTRXINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
4	RXIM: 接收中断屏蔽。读取操作返回 UARTRXINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTRXINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
3	DSRMIM: nUARTDSR 调制解调器中断屏蔽。读取操作返回 UARTDSRINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTDSRINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
2	DCDMIM: nUARTDCD 调制解调器中断屏蔽。读取操作返回 UARTDCDINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTDCDINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
1	CTSMIM: nUARTCTS 调制解调器中断屏蔽。读取返回 UARTCTSINTR 中断的当前屏蔽状态。写入1时, 设置 UARTCTSINTR 中断屏蔽; 写入0时, 清除该屏蔽。	读写	0x0
0	RIMIM: nUARTRI 调制解调器中断屏蔽。读取返回 UARTRIINTR 中断的当前屏蔽状态。写入1时, 设置 UARTRIINTR 中断屏蔽; 写入0时, 清除该屏蔽。	读写	0x0

UART: UARTRIS 寄存器

偏移量: 0x03c

描述

原始中断状态寄存器, UARTRIS

表436. UARTRIS
寄存器

位	描述	类型	复位值
31:11	保留。	-	-

位	描述	类型	复位值
10	OERIS: 溢出错误中断状态。返回UARTOEINTR中断的原始中断状态。	只读	0x0
9	BERIS: 断开错误中断状态。返回UARTBEINTR中断的原始中断状态。	只读	0x0
8	PERIS: 奇偶校验错误中断状态。返回UARTPEINTR中断的原始中断状态。 ◦	只读	0x0
7	FERIS: 帧错误中断状态。返回UARTFEINTR中断的原始中断状态。	只读	0x0
6	RTRIS: 接收超时中断状态。返回UARTRTINTR中断的原始中断状态。	只读	0x0
5	TXRIS: 发送中断状态。返回UARTTXINTR中断的原始中断状态。	只读	0x0
4	RXRIS: 接收中断状态。返回UARTRXINTR中断的原始中断状态。	只读	0x0
3	DSRRMIS: nUARTDSR调制解调器中断状态。返回UARTDSRINTR中断的原始中断状态。	只读	-
2	DCDRMIS: nUARTDCD调制解调器中断状态。返回UARTDCDINTR中断的原始中断状态。	只读	-
1	CTSRMIS: nUARTCTS调制解调器中断状态。返回UARTCTSINTR中断的原始中断状态。	只读	-
0	RIRMIS: nUARTRI调制解调器中断状态。返回UARTRIINTR中断的原始中断状态。	只读	-

UART：UARTMIS寄存器

偏移量: 0x040

描述

掩码中断状态寄存器，UARTMIS

表437. UARTMIS寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEMIS: 溢出错误屏蔽中断状态。返回UARTOEINTR中断的屏蔽中断状态。	只读	0x0
9	BEMIS: 断点错误屏蔽中断状态。返回UARTBEINTR中断的屏蔽中断状态。 ◦	只读	0x0
8	PEMIS: 奇偶校验错误屏蔽中断状态。返回UARTPEINTR中断的屏蔽中断状态。	只读	0x0
7	FEMIS: 帧错误屏蔽中断状态。返回UARTFEINTR中断的屏蔽中断状态。	只读	0x0
6	RTMIS: 接收超时屏蔽中断状态。返回UARTRTINTR中断的屏蔽中断状态。	只读	0x0
5	TXMIS: 发送屏蔽中断状态。返回UARTTXINTR中断的屏蔽中断状态。	只读	0x0

位	描述	类型	复位值
4	RXMIS : 接收屏蔽中断状态。返回UARTRXINTR中断的屏蔽中断状态。	只读	0x0
3	DSRMMIS : nUARTDSR调制解调器屏蔽中断状态。返回UARTDSRINTR中断的屏蔽中断状态。	只读	-
2	DCDMMIS : nUARTDCD调制解调器屏蔽中断状态。返回UARTDCDINTR中断的屏蔽中断状态。	只读	-
1	CTSMMIS : nUARTCTS调制解调器屏蔽中断状态。返回UARTCTSINTR中断的屏蔽中断状态。	只读	-
0	RIMMIS : nUARTRI调制解调器屏蔽中断状态。返回UARTRIINTR中断的屏蔽中断状态。	只读	-

UART: UARTICR寄存器

偏移量: 0x044

描述

中断清除寄存器, UARTICR

表438. UARTICR寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEIC : 溢出错误中断清除。清除UARTOEINTR中断。	WC	-
9	BEIC : 断点错误中断清除。清除UARTBEINTR中断。	WC	-
8	PEIC : 奇偶校验错误中断清除。清除UARTPEINTR中断。	WC	-
7	FEIC : 帧错误中断清除。清除UARTFEINTR中断。	WC	-
6	RTIC : 接收超时中断清除。清除UARTRTINTR中断。	WC	-
5	TXIC : 发送中断清除。清除UARTTXINTR中断。	WC	-
4	RXIC : 接收中断清除。清除UARTRXINTR中断。	WC	-
3	DSRMIC : nUARTDSR 调制解调器中断清除。清除UARTDSRINTR中断 。	WC	-
2	DCDMIC : nUARTDCD 调制解调器中断清除。清除UARTDCDINTR中断 。	WC	-
1	CTSMIC : nUARTCTS 调制解调器中断清除。清除UARTCTSINTR中断 。	WC	-
0	RIMIC : nUARTRI 调制解调器中断清除。清除UARTRIINTR中断。	WC	-

UART: UARTDMACR 寄存器

偏移: 0x048

描述

DMA 控制寄存器, UARTDMACR

表 439。
UARTDMACR 寄存器

位	描述	类型	复位值
31:3	保留。	-	-

位	描述	类型	复位值
2	DMAONERR : DMA错误时。若此位被置为1，当UART错误中断发生时，UART接收请求输出UARTRXDMASREQ或UARTRXDMABREQ将被禁用。	读写	0x0
1	TXDMAE : 传输DMA使能。若此位被置为1，则启用传输FIFO的DMA功能。 ◦	读写	0x0
0	RXDMAE : 接收DMA使能。若此位被置为1，则启用接收FIFO的DMA功能。	读写	0x0

UART：UARTPERIPHID0 寄存器

偏移量：0xfe0

描述

UARTPeriphID0 寄存器

表440。
UARTPERIPHID0
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	PARTNUMBER0 : 这些位的读出值为0x11	只读	0x11

UART：UARTPERIPHID1 寄存器

偏移量：0xfe4

描述

UARTPeriphID1 寄存器

表441。
UARTPERIPHID1
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	DESIGNERO : 这些位的读出值为0x1	只读	0x1
3:0	PARTNUMBER1 : 这些位读取值为 0x0	只读	0x0

UART：UARTPERIPHID2 寄存器

偏移量：0xfe8

描述

UARTPeriphID2 寄存器

表442。
UARTPERIPHID2
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	版本号：该字段取决于UART的版本：r1p0 0x0, r1p1 0x1 r1p3 0x2, r1p4 0x2, r1p5 0x3	只读	0x3
3:0	DESIGNER1 : 这些位读取值为 0x4	只读	0x4

UART：UARTPERIPHID3 寄存器

偏移量：0fec

描述

UARTPeriphID3 寄存器

表 443。
UARTPERIPHID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	配置：这些位读取值为 0x00	只读	0x00

UART：UARTPCELLID0 寄存器

偏移量：0xff0

描述

UARTPCellID0 寄存器

表 444。
UARTPCELLID0
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID0：这些位读取的返回值为 0x0D	只读	0x0d

UART：UARTPCELLID1 寄存器

偏移：0xff4

描述

UARTPCellID1 寄存器

表 445。
UARTPCELLID1
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID1：这些位读取的返回值为 0xF0	只读	0xf0

UART：UARTPCELLID2 寄存器

偏移：0xff8

描述

UARTPCellID2 寄存器

表 446。
UARTPCELLID2
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID2：这些位读取的返回值为 0x05	只读	0x05

UART：UARTPCELLID3 寄存器

偏移：0ffc

描述

UARTPCellID3 寄存器

表 447。
UARTPCELLID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID3：这些位读取的返回值为 0xB1	只读	0xb1

4.3. I2C

Synopsys文档

Synopsys专有，经过授权使用。

I2C是一种常用的两线接口，通过时钟线 **SCL**和数据线 **SDA**连接设备，用于低速数据传输。

RP2040包含两个相同的I2C控制器实例。每个控制器的外部引脚连接至第2.19.2节GPIO复用表中定义的GPIO引脚，该复用选项提供了一定的输入输出灵活性。

4.3.1. 特性

每个I2C控制器基于Synopsys DW_apb_i2c (v2.01) IP配置，支持以下功能：

- 主设备模式或从设备模式（默认为主设备模式）
- 支持标准模式、快速模式及快速增强模式
- 默认从设备地址为0x055
- 主设备模式支持10位地址
- 16单元传输缓存
- 16单元接收缓存
- 支持由DMA驱动
- 支持中断生成

4.3.1.1. 标准

I2C控制器依据2014年4月发布的I2C总线规范6.0版设计。

4.3.1.2. 时钟

I2C控制器的所有时钟均连接至 **clk_sys**，包括后文所述的 **ic_clk**。I2C时钟通过分频该时钟产生，由模块内部寄存器控制。

4.3.1.3. 输入/输出

每个控制器必须将其时钟线 **SCL**和数据线 **SDA**连接至一对GPIO。I2C标准要求驱动器主动拉低信号，若未驱动信号则自动被上拉。此要求适用于SCL和SDA。GPIO引脚应配置为：

- 启用上拉
- 限制转换速率
- 启用施密特触发器

● 注意

板上应配备外部上拉电阻，因为内部引脚的上拉能力可能不足以满足外部电路的上拉需求。

4.3.2. I₂C 配置

I₂C 配置细节（每个实例完全独立）：

- 32 位 APB 访问
- 支持标准模式、快速模式或快速模式增强版（不支持高速模式）
- 默认从机地址为 0x055
- 主模式或从模式
- 默认主模式（复位时从模式禁用）
- 主模式支持 10 位寻址（默认 7 位）
- 16 项传输缓冲区
- 16 项接收缓冲区
- 允许主机发起重启条件（可为兼容旧设备而禁用）
- 可配置时序以调整 TsuDAT/ThDAT
- 复位时响应广播调用
- DMA 接口
- 单一中断输出
- 可配置时序以调整时钟频率
- 尖峰抑制（默认 7 个 clk_sys 周期）
- 从机接收数据后可发送 NACK
- 当 TX FIFO 为空时保持传输
- 保持总线，直到 RX FIFO 中有可用空间
- 从模式下重新检测中断
- 可选阻塞式主机命令（默认未启用）

4.3.3. I₂C 概述

I₂C 总线是一种由串行数据线 SDA 和串行时钟线 SCL 组成的双线串行接口。这些线路在总线上连接的设备之间传递信息。每个设备以唯一地址识别，且可根据设备功能作为“发送方”或“接收方”操作。设备在执行数据传输时，也可以被视为主设备或从设备。主设备是指启动总线数据传输并生成时钟信号以允许该传输的设备。此时，任何被寻址的设备均视为从设备。

注意

I2C模块必须仅被编程为主模式或从模式中的一种。不支持主从并存的操作模式。

I2C模块可在以下模式下运行：

- 标准模式（数据速率范围0至100kbps），
- 快速模式（数据传输速率小于或等于400kbps），
- 快速增强模式（数据传输速率小于或等于1000kbps）。

以下模式不受支持：

- 高速模式（数据传输速率小于或等于3.4Mbps），
- 超高速模式（数据传输速率小于或等于5Mbps）。

注意

对快速模式的提及同样适用于快速增强模式，除非另有明确说明。

只要设备连接到总线，I2C模块即可与这些模式的设备通信。

此外，快速模式设备具有向下兼容性。例如，快速模式设备能够在0至100kbps的I2C总线系统中与标准模式设备通信。然而，标准模式设备不具备向上兼容性，不应纳入快速模式I2C总线系统中，因为其无法适应更高的传输速率，可能导致不可预测的状态。

高速模式设备的示例包括液晶显示器、高位数模数转换器和大容量EEPROM。这些设备通常需传输大量数据。大多数维护和控制应用程序，即I2C总线的常见用途，通常在100kHz频率（标准模式和快速模式）下运行。任何DW_apb_i2c设备均可连接至I2C总线，且每个设备均可与任意主设备通信，实现信息的双向传输。总线上必须至少有一个主设备（如微控制器或数字信号处理器），但亦可存在多个主设备，需通过仲裁机制确定总线所有权。多个主设备及仲裁机制将在本章后续部分详细阐述。I2C模块不支持SMBus及PMBus协议（分别用于系统管理与电源管理）。

DW_apb_i2c由AMBA APB从属接口、I2C接口及保持两者一致性的FIFO逻辑组成。组件各模块结构如图64所示。

图64. I2C块图



以下定义图64各模块的功能：

- **AMBA总线接口单元**— 将APB接口信号转换为通用接口，使寄存器文件与总线协议无关。

- **寄存器文件**—含配置寄存器，是与软件的接口。
- **从状态机**—遵循从机协议，监控总线地址匹配。
- **主状态机**—生成主机传输所需的I2C协议。
- **时钟发生器**—计算以下操作所需的时序：
 - 配置为主设备时生成 **SCL**时钟
 - 检查总线空闲状态
 - 生成START和STOP信号
 - 设置并保持数据
- **Rx Shift** — 将数据导入设计并以字节格式提取。
- **Tx Shift** — 呈现CPU提供的数据以便在I2C总线上传输。
- **Rx Filter** — 检测总线上的事件；例如起始信号、停止信号及仲裁丢失。
- **Toggle** — 在两端产生脉冲并切换，用以跨时钟域传输信号。
- **Synchronizer** — 将信号从一个时钟域传输至另一时钟域。
- **DMA Interface**—向中央DMA控制器生成握手信号，实现数据传输自动化，无需CPU介入。
- **Interrupt Controller** — 生成原始中断信号与中断标志，允许设置及清除。
- **RX FIFO/TX FIFO** — 保持RX FIFO和TX FIFO寄存器组及其控制器，含状态级别。

4.3.4. I2C 术语

以下术语定义如下：

4.3.4.1. I2C总线术语

以下术语涉及I2C设备的角色及其与总线上的其他I2C设备的交互方式。

- **发送器**—向总线发送数据的设备。发送器可以是主动发起数据传输的设备（主发送器），也可以是响应主设备请求向总线发送数据的设备（从发送器）。
- 接收器—从总线接收数据的设备。接收器可以是自行请求接收数据的设备（主接收器），也可以是响应主设备请求的设备（从接收器）。
- 主设备—初始化传输（START命令）、生成时钟信号**SCL**并终止传输（STOP命令）的组件。主设备可以是发送器或接收器。
- 从设备—由主设备寻址的设备。从设备可以是接收器或发送器。
- **多主设备**—指总线上可同时存在多个主设备且不会发生冲突或数据丢失的能力。
- **仲裁**—指预定义程序，授权每次仅允许一个主设备控制总线。有关此行为的更多信息，参见第4.3.8节。
- **同步**—指预定义程序，用于同步两个或多个主设备所提供的时钟信号。
有关此功能的更多信息，参见第4.3.9节。
- **SDA** — 数据线（串行数据）
- **SCL** — 时钟线（串行时钟）

4.3.4.2. 总线传输术语

以下术语专指发生于I2C总线上的数据传输。

- 启动（重新启动） – 数据传输以启动或重新启动条件开始。SDA数据线电平由高变为低，同时SCL时钟线保持高电平。当此情况发生时，总线处于忙碌状态。

i 注意

START 和 RESTART 条件在功能上完全相同。

- STOP – 通过 STOP 条件终止数据传输。当 SDA 数据线电平由低变高，而 SCL 时钟线保持高电平时，即发生此现象。数据传输终止后，总线重新处于空闲状态。若生成 RESTART 而非 STOP 条件，总线将保持忙碌状态。

4.3.5. I2C 行为

DW_apb_i2c 可通过软件控制，设定为以下任一模式：

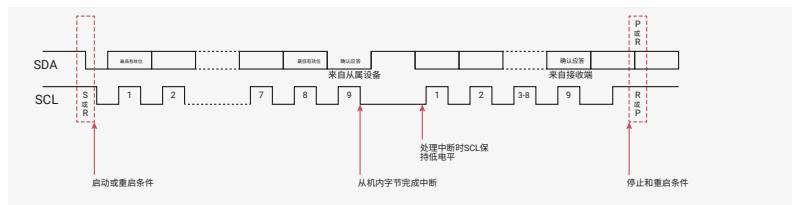
- 仅作为 I2C 主设备，与其他 I2C 从设备通信；或
- 仅作为 I2C 从设备，与一个或多个 I2C 主设备通信。

主设备负责产生时钟并控制数据传输。从设备负责向主设备发送或接收数据。数据确认由接收设备发出，该设备可为主设备或从设备。如前所述，I2C 协议亦允许多个主设备共存于 I2C 总线上，并通过仲裁程序来确定总线所有权。

每个从设备均拥有由系统设计者确定的唯一地址。当主设备欲与从设备通信时，主设备首先发送 START/RESTART 条件，随后发送从设备地址及控制位 (R/W)，以确定主设备是欲发送数据至从设备，还是从从设备接收数据。从设备接收地址后，会发送应答 (ACK) 信号。

若主设备（主发送方）向从设备（从接收方）写入数据，接收方将接收一个字节的数据。此传输过程持续进行，直到主设备通过发送 STOP 条件终止传输。若主设备从从设备读取数据（主接收方），从设备（从发送方）向主设备发送一个字节数据，主设备随后通过 ACK 信号确认该传输。此传输过程持续进行，直至主设备在接收最后一个字节后不再发送 ACK 信号（即发送 NACK），随后主设备发送 STOP 条件终止传输，或在发送 RESTART 条件后寻址另一从设备。该行为如图 65 所示。

图 65. I2C 总线上的数据传输



DW_apb_i2c 为同步串行接口。SDA 线为双向信号，仅在 SCL 线为低电平时发生变化，停止 (STOP)、起始 (START) 及重新起始 (RESTART) 条件除外。输出驱动为开漏或开集电极，以在总线上实现线与 (wire-AND) 功能。总线上设备的最大数量仅受最大电容规格 400 pF 的限制。数据以字节包形式传输。

DW_apb_i2c 中实现的 I2C 协议详见第 4.3.6 节。

4.3.5.1. 起始 (START) 与停止 (STOP) 信号的产生

作为 I2C 主设备时，向传输 FIFO 写入数据会触发 DW_apb_i2c 在 I2C 总线上生成起始 (START) 信号。向 IC_DATA_CMD.ST_OP 写入 1 会使 DW_apb_i2c 在

I2C总线上生成停止（STOP）信号；若未设置此位，即使传输FIFO为空，也不会发出停止（STOP）信号。

作为从设备运行时，DW_apb_i2c 不会产生协议中的 START 和 STOP 条件。

但如果向 DW_apb_i2c 发出读请求，其会将 SCL 线保持低电平，直到向其提供读取数据。

这将使 I2C 总线阻塞，直到向从设备 DW_apb_i2c 提供读取数据，或通过向 IC_ENABLE.ENABLE 寄存器写入 0 禁用 DW_apb_i2c 从设备。

4.3.5.2. 组合格式

DW_apb_i2c 支持 7 位和 10 位寻址模式下的混合读写组合格式事务。DW_apb_i2c 不支持混合地址及混合地址格式的组合格式事务，即先进行 7 位地址事务后再进行 10 位地址事务，或反之。要启动组合格式传输，应将 IC_CON.IC_RESTART_EN 设置为 1。当该值设置且作为主设备运行时，DW_apb_i2c 在完成一次I2C传输后，会检查发送FIFO并执行下一次传输。如果此次传输方向与上一次传输不同，则采用组合格式发起该传输。当当前I2C传输完成且发送FIFO为空时：

- 将检查IC_DATA_CMD.STOP位，且：
 - 若该位设为1，则发送STOP位。
 - 若该位设为0，则 SCL 线保持低电平，直到下一条命令写入发送FIFO。

详细信息请参见第4.3.7节。

4.3.6. I2C 协议

DW_apb_i2c 支持本节所述的协议。

4.3.6.1. START与STOP条件

当总线处于空闲状态时，SCL 和 SDA 信号线通过外部上拉电阻被拉高。当主设备欲在总线上开始传输时，主设备发出 START 条件。定义为在 SCL 为 1 时，SDA 信号由高电平到低电平的跳变。当主设备欲终止传输时，主设备应发出 STOP 条件。定义为在 SCL 为 1 时，SDA 线路由低电平跳变至高电平。图66显示了 START 和 STOP 条件的时序。当数据在总线上传输时，SCL 为 1 期间，SDA 线路必须保持稳定。

图66。I2C START 和 STOP 条件



① 注意

图66所示的START/STOP条件信号跳变，反映了主设备驱动I2C总线时的输出信号。观察从设备输入端的 SDA/SCL 信号时须谨慎，因线路延迟不一致可能导致 SDA/SCL 时序关系错误。

4.3.6.2. 从设备协议寻址

地址格式分为两种：7位地址格式和10位地址格式。

4.3.6.2.1. 7位地址格式

在7位地址格式中，首字节的高七位（第7至第1位）表示从设备地址，最低位（第0位）为读/写标志位，如图67所示。当第0位（读/写）为0时，主设备向从设备写入数据。当第0位（读/写）为1时，主设备从从设备读取数据。

图67. I_C 7位地址格式



4.3.6.2.2. 10位地址格式

在10位寻址过程中，传输两个字节以设置10位地址。第一个字节的传输内容包括以下位定义。前五位（位7至位3）通知从设备此次操作作为10位寻址，随后两位（位2至位1）设置从设备地址的第9位和第8位，最低有效位（位0）为读写位。第二个传输字节设置从设备地址的位7至位0。图68展示了10位地址格式。

图68. 10位地址格式



本表定义了特殊用途及保留的首字节地址。

表448。I_C/SMBus首字节各位定义

从设备地址	读写位	描述
0000 000	0	通用呼叫地址。DW_apb_i2c将数据置于接收缓冲区并触发通用调用中断。
0000 000	1	START 字节。更多详情，请参见第4.3.6.4节。
0000 001	X	CBUS 地址。DW_apb_i2c 忽略此类访问。
0000 010	X	保留。
0000 011	X	保留。
0000 1XX	X	高速主机代码（详见第4.3.8节）。
1111 1XX	X	保留。
1111 0XX	X	10位从机地址。
0001 000	X	SMBus 主机（不支持）
0001 100	X	SMBus 警报响应地址（不支持）
1100 001	X	SMBus 设备默认地址（不支持）

DW_apb_i2c 不限制您使用这些保留地址。但若您使用这些保留地址，

可能导致与其他 I2C 组件的不兼容。

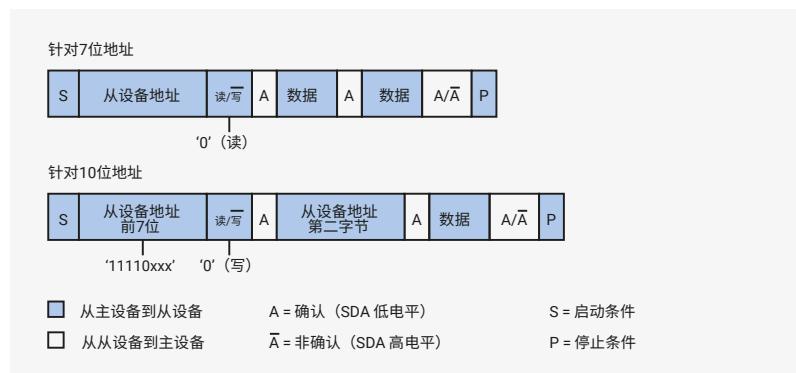
4.3.6.3 传输与接收协议

主机可发起总线上的数据传输与接收，既可作为主发送器，也可作为主接收器。从设备响应主设备的请求，执行数据的发送或接收操作，分别作为从设备发送方或从设备接收方。

4.3.6.3.1. 主设备发送方与从设备接收方

所有数据均以字节格式传输，单次数据传输的字节数无限制。主设备发送地址和读写位后，或主设备向从设备传输一个字节数据后，从设备接收方必须发送应答信号（ACK）。当从设备接收方未响应ACK信号时，主设备通过发送停止条件终止传输。从设备必须保持 SDA 线高电平，以便主设备能够终止传输。如果主设备发送方如图69所示传输数据，则从设备接收方在每接收一个字节数据后向主设备发送应答信号。

图69。I2C主控发送协议



4.3.6.3.2. 主接收器与从传输器

如图70所示，若主设备正在接收数据，则在接收完每个数据字节后（除最后一个字节外），主设备会向从传输器发送确认脉冲。这是主接收器通知从发送器这是最后一个字节的方式。从发送器在检测到无应答（NACK）后释放 SDA线，以便主设备发出停止条件。

图70。I2C主设备-接收器协议



当主设备不希望通过停止条件释放总线时，可发出重启条件。

此条件与起始条件相同，但发生在应答脉冲之后。在主模式下运行时，DW_apb_i2c可通过方向不同的传输与同一从设备通信。有关DW_apb_i2c支持的复合格式事务的描述，请参见第4.3.5.2节。

注意

在重新编程目标从设备地址寄存器 (IC_TAR) 之前，必须完全禁用 DW_apb_i2c。

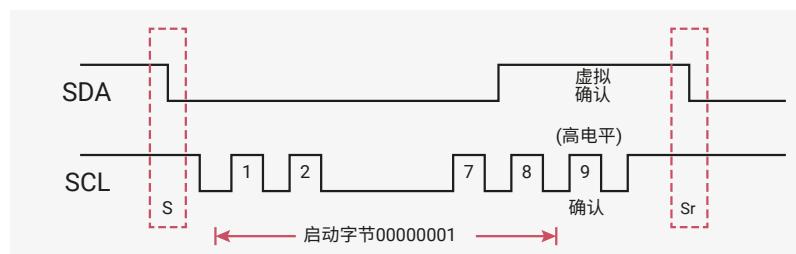
4.3.6.4. START BYTE 传输协议

START BYTE 传输协议适用于无板载专用 I2C 硬件模块的系统。

当 DW_apb_i2c 作为从设备被寻址时，始终以支持的最高速度采样 I2C 总线，因此不需要 START BYTE 传输。但当 DW_apb_i2c 作为主设备时，支持在每次传输开始时生成 START BYTE 传输，以应对从设备的需求。

该协议包括先发送七个零位，随后发送一个一位，如图 71 所示。这使得轮询总线的处理器可在地址阶段低速采样，直到检测到零位。检测到零位后，微控制器将采样速率从低速切换至主设备的正确速率。

图 71. I2C 启动字节传输



START BYTE 过程如下：

1. 主机生成START起始条件。
2. 主机发送START字节 (0000 0001)。
3. 主机发送ACK应答时钟脉冲。（仅为符合总线上字节处理格式而存在）
4. 无任何从机将ACK信号置零。
5. 主机生成RESTART（重复启动）条件。

硬件接收器不会响应START字节，因为该地址被保留，并在生成RESTART条件后复位。

4.3.7. 发送 FIFO 管理及 START、STOP 和 RESTART 生成

作为主机操作时，DW_apb_i2c 组件支持图 72 中所示的 Tx FIFO 管理模式。

4.3.7.1. Tx FIFO管理

当 Tx FIFO 变为空时，组件不会生成 STOP 条件；此时组件将 SCL 线保持低电平，暂停总线，直到 Tx FIFO 中有新数据。仅当用户通过设置写入 IC_DATA_CMD 寄存器命令的第九位（停止位）时，才会生成 STOP 条件。图 72 显示了 IC_DATA_CMD 寄存器中的各个位。

图72。
IC_DATA_CMD
寄存器



图73说明了 DW_apb_i2c 作为主机发送器时，发送FIFO（Tx FIFO）为空时的行为，以及停止（STOP）条件的生成。

图73。主控
发送器 - Tx FIFO
清空/STOP
生成

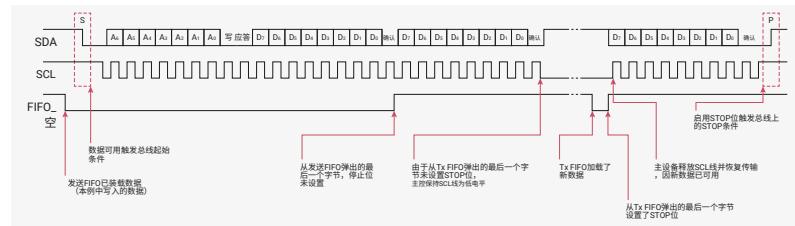


图74展示了DW_apb_i2c作为主控接收器时，Tx FIFO变为空的行为及STOP条件的生成。

图 74。主机
接收器 - 发送 FIFO
清空/停止
生成

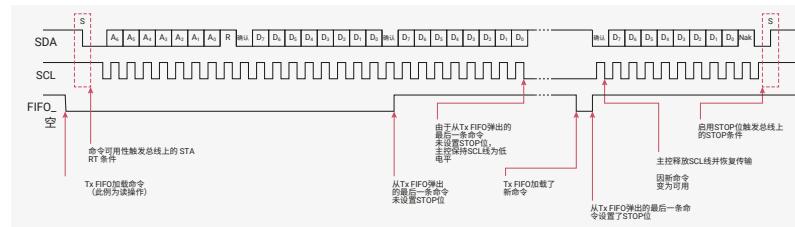


图 75 和图 76 展示了用户可控制 I2C 总线上 RESTART 条件生成的配置。如果寄存器 IC_DATA_CMD 中的第 10 位 (Restart) 被设置，且重启功能已启用 (IC_RESTART_EN=1)，则在数据字节写入或读取从机之前将生成 RESTART。如果未启用重启功能，则会先生成 STOP，随后生成 START 以代替 RESTART。图 75 展示了作为主机发送方在此情况下的操作。

图 75。主机
发送方 - 重启
IC_DATA_CMD 寄存器
的位被设置

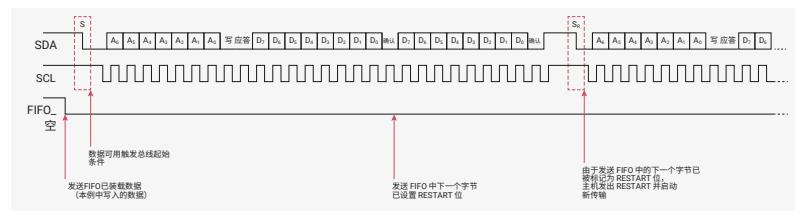


图76展示了作为主设备接收器时的相同情况。

图76。主接收
器--IC_DATA_CMD
寄存器的Restart
位已设置

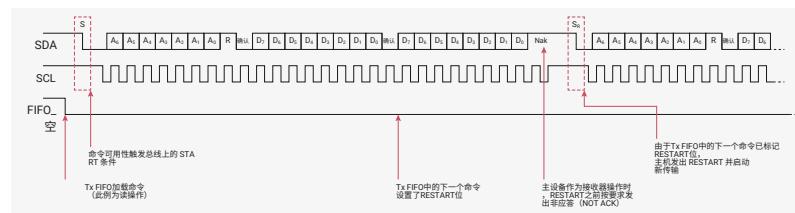


图77展示了作为主设备发送器时，IC_DATA_CMD寄存器Stop位设置且Tx FIFO非空的操作

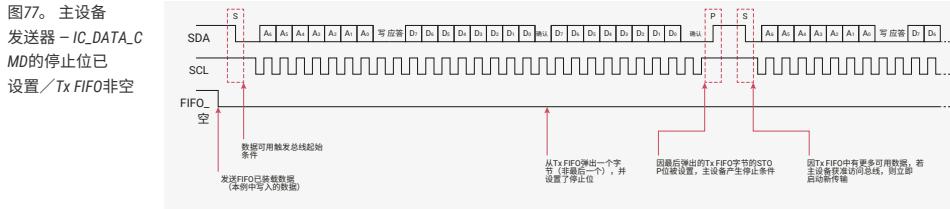


图77展示了主设备发送器 - IC_DATA_C 的时序图

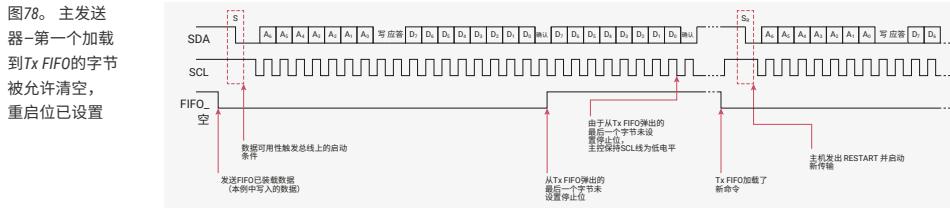


图78展示了作为主设备发送器时，加载至 Tx FIFO 的首字节在Restart位被设置的情况下允许清空的操作

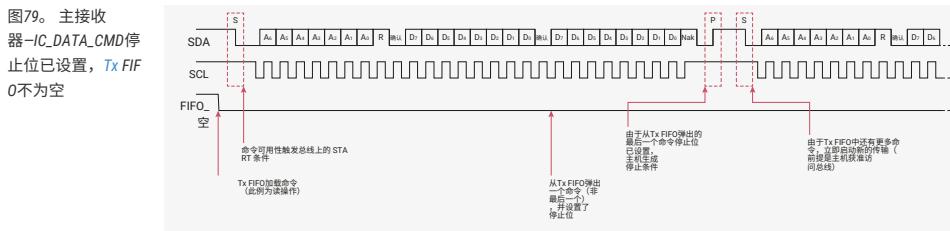
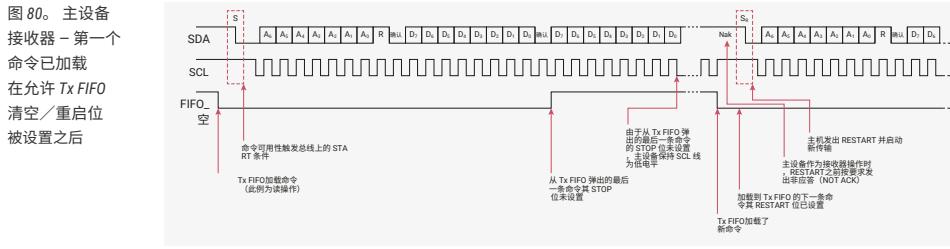


图79展示了主接收器的操作，其中 IC_DATA_CMD 寄存器的停止位已设置且 Tx FIFO 不为空



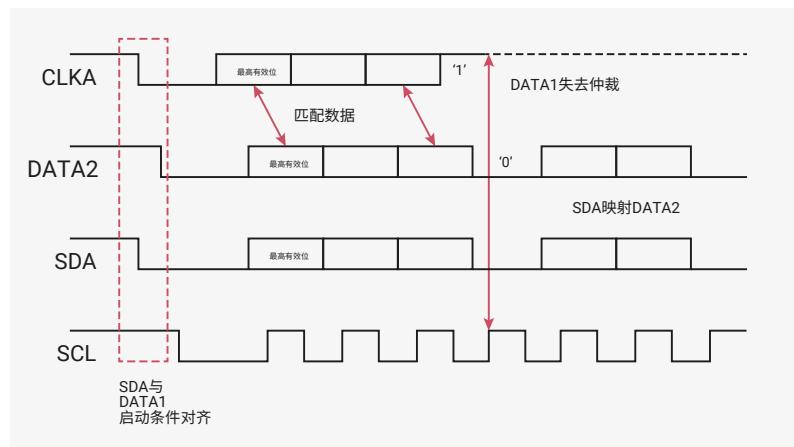
4.3.8. 多主机仲裁

DW_apb_i2c 总线协议允许多个主设备共存于同一总线上。当同一 I2C 总线上存在两个主设备且它们同时尝试生成 START 条件以控制总线时，将启动仲裁流程。一旦某一主设备（例如微控制器）获得总线控制权，其他主设备在该主设备发送 STOP 条件并使总线处于空闲状态之前，均无法获得控制权。

仲裁过程发生在 SDA 线上，而 SCL 线保持为高电平。发送“1”信号的主设备在另一主设备发送“0”时失去仲裁权，并关闭其数据输出阶段。失去仲裁的主设备可以继续生成时钟，直至字节传输结束。若两个主设备均寻址同一从设备，仲裁可能进入数据阶段。

检测到被另一主设备仲裁失败后，DW_apb_i2c 将停止生成 SCL（禁用输出驱动）。图81展示了两个主设备在总线上进行仲裁的时序。

图81。多主设备
仲裁



总线控制由竞争主设备发送的地址或主代码及数据决定，因此总线上不存在中央主设备或优先级顺序。

如下情况不允许发生仲裁：

- 重新启动（RESTART）条件与数据位
- 停止（STOP）条件与数据位
- 重新启动（RESTART）条件与停止（STOP）条件

① 注意

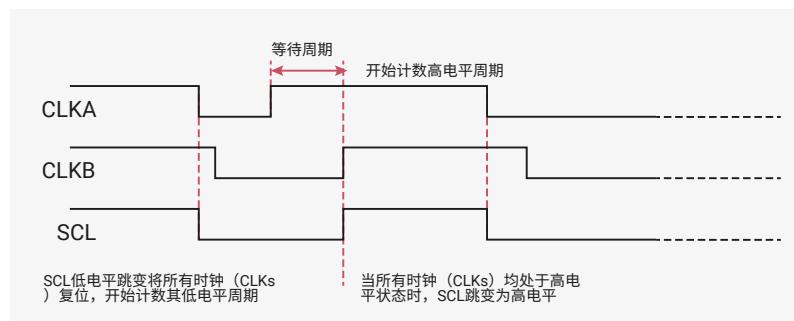
从设备不参与仲裁过程。

4.3.9. 时钟同步

当两个或多个主设备同时尝试在总线上传输信息时，必须对 SCL时钟进行仲裁和同步。所有主设备均生成各自的时钟以传输消息。数据仅在 SCL时钟的高电平期间有效。时钟同步通过对 SCL信号的有线与（wired-AND）连接实现。当主设备将 SCL时钟拉低至0时，开始计时 SCL时钟的低电平时间，并在下一时钟周期开始时将 SCL时钟信号置高。但如果另一主设备持续将 SCL线路保持为0，则该主设备将进入高电平等待状态，直至 SCL时钟线路变为1。

所有主设备随后计数其高电平时间，最短高电平时间的主设备将 SCL线路拉低。然后主设备计数其低电平时间，最长低电平时间的主设备强制其它主设备进入高电平等待状态。因此生成了一个同步的 SCL时钟，如图82所示。从设备可选择将SCL线拉低，以减缓I2C总线上的时序。

图82。多主时
钟同步



4.3.10. 操作模式

本节提供有关操作模式的信息。

i 注意

务必注意，DW_apb_i2c应仅配置为I2C主机或I2C从机，不得同时兼备两者。通过确保IC_CON.IC_SLAVE_DISABLE不设置为零且IC_CON.MASTER_MODE不设置为一，达到此目的。

4.3.10.1 从机模式操作

本节讨论从机模式的操作程序。

4.3.10.1.1 初始配置

要将DW_apb_i2c用作从机，请执行以下步骤：

1. 通过向 IC_ENABLE.ENABLE 寄存器写入‘0’来禁用 DW_apb_i2c。
 2. 向 IC_SAR 寄存器（位 9:0）写入以设置从机地址。该地址为 DW_apb_i2c 响应的地址。
 3. 向 IC_CON 寄存器写入以指定支持的寻址类型（通过设置第3位选择7位或10位寻址）。
- 通过向第六位（IC_SLAVE_DISABLE）写入‘0’，以及向第零位（MASTER_MODE）写入‘0’，启用 DW_apb_i2c 的仅从机模式。

i 注意

从机和主机可以使用不同的寻址类型，支持7位或10位地址。例如，主机可设置为10位寻址，从机可设置为7位寻址，反之亦然。

1. 通过向 IC_ENABLE.ENABLE 寄存器写入‘1’来启用 DW_apb_i2c。

i 注意

根据复位时选定的初始值，步骤二和三可能不必执行，因为复位值可被配置。例如，如果设备仅用作主设备，则无需设置从设备地址，因为可以配置 DW_apb_i2c 在复位后禁用从设备，并在复位后启用主设备。

存储的值为静态值，若 DW_apb_i2c 被禁用，则无需重新编程。

— 警告

建议仅在 I2C 总线处于空闲（IDLE）状态时，才将 DW_apb_i2c 从复位状态释放。若在传输过程中取消复位，内部用于同步 SDA 和 SCL 的同步触发器将从复位值“1”切换为总线上的实际值，从而可能导致在 SCL 为“1”时，SDA 从“1”切换为“0”，使 DW_apb_i2c 从设备误判为虚假起始条件。该情况亦可通过配置 DW_apb_i2c，将 IC_SLAVE_DISABLE 设为 1 且 MASTER_MODE 设为 1，以在复位后禁用从设备接口来避免。随后，通过编程设置 IC_CON[0] = 0 和 IC_CON[6] = 0 启用该功能，前提是内部 SDA 和 SCL 已与总线上的信号同步；此过程大约需要六个

ic_clk 周期，发生于复位撤销之后。

4.3.10.1.2. 单字节从设备发送操作

当总线上另一I2C主设备寻址DW_apb_i2c并请求数据时，DW_apb_i2c作为从设备发送器，执行以下步骤：

1. 另一I2C主设备发起的I2C传输，其地址与DW_apb_i2c的IC_SAR寄存器中设置的从设备地址匹配。
2. DW_apb_i2c确认所发送地址，并识别传输方向，表明其作为从设备发送器运行。
3. DW_apb_i2c触发RD_REQ中断（IC_RAW_INTR_STAT寄存器第5位），并将SCL线拉低，进入等待状态，直至软件响应。如果由于IC_INTR_MASK.M_RD_REQ被设置为零而导致RD_REQ中断被屏蔽，建议使用硬件和/或软件定时程序，指示CPU定期读取IC_RAW_INTR_STAT寄存器。
 - a. 将IC_RAW_INTR_STAT.RD_REQ设为一的读取操作必须视同RD_REQ中断被触发。
 - b. 软件随后必须采取措施以完成I2C传输。
 - c. 所使用的定时间隔应约为DW_apb_i2c可处理的最快SCL时钟周期的十倍。例如，对于400kbps，定时间隔为25μs。

i 注意

此处推荐使用数值10，原因是这大致相当于I2C总线上传输单个字节所需的时间。

1. 如果在收到读取请求之前，Tx FIFO中仍有数据残留，则DW_apb_i2c会触发TX_ABRT中断（IC_RAW_INTR_STAT寄存器的第六位）以清空Tx FIFO中的旧数据。若由于IC_INTR_MASK.M_TX_ABRT被设置为零而导致TX_ABRT中断被屏蔽，建议重复使用前述定时程序或类似程序，读取IC_RAW_INTR_STAT寄存器。

i 注意

由于每当发生TX_ABRT事件时，DW_apb_i2c的Tx FIFO会被强制刷新或重置，软件必须通过读取IC_CLR_TX_ABRT寄存器以释放DW_apb_i2c的该状态，然后方可尝试向Tx FIFO写入数据。详见寄存器IC_RAW_INTR_STAT。

- a. 当读取中第六位（R_TX_ABRT）被置为1时，应视为TX_ABRT中断已被断言。
- b. 软件无需采取进一步操作。
- c. 所使用的时间间隔应与前一步中对IC_RAW_INTR_STAT.RD_REQ寄存器描述的时间间隔类似。
 1. 软件通过向IC_DATA_CMD寄存器写入数据（将第8位写为'0'）实现写操作。
 2. 软件在继续执行前，必须清除IC_RAW_INTR_STAT寄存器中的RD_REQ和TX_ABRT中断（分别对应第5位和第6位）。如果RD_REQ和/或TX_ABRT中断已被屏蔽，则当读取到R_RD_REQ或R_TX_ABRT位为1时，系统将已完成对IC_RAW_INTR_STAT寄存器的清除。
 3. DW_apb_i2c释放SCL并传输字节。
 4. 主设备可通过发出RESTART条件保持I2C总线，或通过发出STOP条件释放总线。

注意

单字节从机发送操作在超高速模式下不适用，因为该模式不支持读取传输。

4.3.10.1.3 单字节从机接收操作

当总线上另一I2C主设备寻址DW_apb_i2c并发送数据时，DW_apb_i2c作为从机接收者，执行以下步骤：

1. 另一I2C主设备使用与DW_apb_i2c在IC_SAR寄存器中所设从机地址匹配的地址启动I2C传输。
2. DW_apb_i2c确认所发送的地址，并识别传输方向，从而表明其作为从机接收者。
3. DW_apb_i2c接收传输的字节并存入接收缓冲区。

注意

如果在推送字节时，Rx FIFO 已完全填满数据，则 DW_apb_i2c 从机会将 I2C SCL 线拉低，直到 Rx FIFO 有可用空间，然后继续执行下一个读取请求。

1. DW_apb_i2c 会触发 RX_FULL 中断 IC_RAW_INTR_STAT.RX_FULL。若因将 IC_INTR_MASK.M_RX_FULL 寄存器设为零或将 IC_TX_TL 设为大于零的值而屏蔽了 RX_FULL 中断，建议实现一个时序例程（详见第 4.3.10.1.2 节）以周期性读取 IC_STATUS 寄存器。软件读取 IC_STATUS 寄存器时，若第 3 位 (RFNE) 为 1，应将该状态视为 RX_FULL 中断已被触发的等效状态。
2. 软件可从 IC_DATA_CMD 寄存器 (位 7:0) 读取该字节。
3. 另一主设备可能通过发出 RESTART 条件来占用 I2C 总线，或通过发出 STOP 条件释放总线。

4.3.10.1.4. 大批量传输的从设备传输操作

在标准I2C协议中，所有事务均为单字节传输，程序通过向从设备的TX FIFO写入一个字节以响应远程主机的读取请求。当从设备（从机发送器）收到远程主机（主机接收器）发出的读取请求 (RD_REQ) 时，TX FIFO中至少应存放一条数据。DW_apb_i2c设计支持在TX FIFO中存储更多数据，以使后续读取请求能够获取这些数据，而无需触发中断以获取更多数据。此举最终消除了若仅允许TX FIFO中存放一条数据而每次获取数据均触发中断所可能导致的显著延迟。此模式仅在DW_apb_i2c作为从机发送器时发生。若远程主机确认了从机发送器发送的数据且从机的TX FIFO中无数据，DW_apb_i2c将在触发读取请求中断 (RD_REQ) 时将I2C SCL线维持低电平，并等待数据写入TX FIFO后方能发送至远程主机。

如果由于 IC_INTR_STAT.R_RD_REQ 被置零而导致 RD_REQ 中断被屏蔽，建议使用定时例程周期性激活对 IC_RAW_INTR_STAT 寄存器的读取。对 IC_RAW_INTR_STAT 的读取返回第5位 (RD_REQ) 为1时，必须视同本节所述的 RD_REQ 中断。该定时例程与第4.3.10.1.2节中描述的类似。

RD_REQ 中断在读请求发生时触发，且须在中断服务例程 (ISR) 退出时清除，如同其他中断。ISR 允许写入一个或多个字节至 Tx FIFO。在向主设备传输这些字节期间，若主设备确认了最后一个字节，则从设备必须再次触发 RD_REQ，因为主设备请求更多数据。若程序员事先已知远程主机请求 ‘n’ 字节的数据包，则当其他主机访问 DW_apb_i2c 并请求数据时，可向 Tx FIFO 写入 ‘n’ 字节，远程主机会连续接收该数据流。例如，

只要远程主控确认已接收所发送的数据且Tx FIFO中仍有可用数据，DW_apb_i2c从设备将持续向远程主控发送数据，无需将SCL线保持为低电平或再次发出RD_REQ信号。

若远程主控预期从DW_apb_i2c接收‘n’字节数据，但程序员向Tx FIFO写入字节数大于‘n’，则从设备在发送完请求的‘n’字节后，会清空Tx FIFO并忽略多余字节。

在此示例中，DW_apb_i2c会产生传输中止（TX_ABRT）事件，以指示已清空Tx FIFO。当预期接收ACK/NACK时，若接收到NACK，则表明远程主控已获取全部所需数据。此时，从设备状态机内部的标志位被激活，用于清除Tx FIFO中剩余的数据。该标志位会传递至包含该FIFO的处理器总线时钟域，在该时钟域中执行清空操作。

4.3.10.2. 主机模式操作

本节讨论主机模式的操作流程。

4.3.10.2.1. 初始配置

若需将DW_apb_i2c用作主机，请执行以下步骤：

1. 通过向IC_ENABLE.ENABLE写入零以禁用DW_apb_i2c。
2. 写入IC_CON寄存器，设置支持的最高速度模式（第2至1位）及所需的DW_apb_i2c主机启动传输速度，同时设置7位或10位寻址模式（第4位）。确保第六位（IC_SLAVE_DISABLE）写入‘1’，第零位（MASTER_MODE）写入‘1’。

注意：从机与主机的地址类型（7位或10位）不必相同。例如，主机可设置为10位寻址，从机可设置为7位寻址，反之亦然。

1. 向IC_TAR寄存器写入要寻址的I2C设备地址（第9至0位）。该寄存器还指示I2C是否将执行通用调用（General Call）或启动字节（START BYTE）命令。
2. 通过向IC_ENABLE.ENABLE位写入1来启用DW_apb_i2c。
3. 现在将传输方向和要发送的数据写入IC_DATA_CMD寄存器。如果在启用DW_apb_i2c之前写入IC_DATA_CMD寄存器，则数据和命令将丢失，因DW_apb_i2c禁用时缓冲区会保持清空状态。此步骤在DW_apb_i2c上产生启动条件和地址字节。一旦启用DW_apb_i2c且TX FIFO中有数据，DW_apb_i2c便开始读取数据。

注意

根据选择的复位值，步骤二、三、四和五可能不必执行，因为复位值可被配置。所存储的值为静态值，除传输方向和数据外，即使DW_apb_i2c被禁用也无需重新编程。

4.3.10.2.2. 主设备发送与主设备接收

DW_apb_i2c 支持在读写之间动态切换。要传输数据，请将待写入的数据写入 I2C 接收/发送数据缓冲区和命令寄存器（IC_DATA_CMD）的低字节。对于 I2C 写操作，CMD 位[8]应写为零。随后，可通过向 IC_DATA_CMD 寄存器低字节写入“无关数据”，并将 CMD 位写为一来发出读命令。只要发送 FIFO 中存在命令，DW_apb_i2c 主设备将持续发起传输。若发送 FIFO 变为空，主设备将在完成当前传输后插入停止（STOP）条件。

- 若设置为 1，则在完成当前传输后发出停止（STOP）条件。
- 若设置为 0，则保持 SCL 线处于低电平，直到下一条命令写入发送 FIFO。

详细信息请参见第4.3.7节。

4.3.10.3. 禁用 DW_apb_i2c

添加了寄存器 IC_ENABLE_STATUS，使软件能够明确判断硬件在 IC_ENABLE.ENABLE 从 1 变为 0 后何时完全关闭。

只需监控一个寄存器，而非早期版本 DW_apb_i2c 需要监控的两个寄存器（IC_STATUS 和 IC_RAW_INTR_STAT）。

i 注意

只有当 ic_enable 取消断言时，当前正在处理的命令的 STOP 位被置为 1，DW_apb_i2c 主控器才能被禁用。若尝试在处理未设置 STOP 位的命令时禁用 DW_apb_i2c 主控器，则主控器将保持激活状态，并保持 SCL 线路为低电平，直到 Tx FIFO 中接收到新的命令。处理未设置 STOP 位的命令时，可以通过发出 ABORT (IC_ENABLE.ABORT) 指令释放 I2C 总线，然后禁用 DW_apb_i2c。

4.3.10.3.1. 操作流程

1. 定义一个计时器间隔 (t_{i2c_poll})，其等于系统中使用且由DW_apb_i2c支持的最高I2C传输速率的信号周期的10倍。例如，若最高I2C传输模式为400kbps，则 t_{i2c_poll} 为25μs。
2. 定义最大超时参数MAX_T_POLL_COUNT，若任何重复轮询操作超过该最大值，则报告错误。
3. 执行阻塞线程/进程/函数，阻止软件启动任何新的I2C主设备事务，但允许正在进行的传输完成。

i 注意

若DW_apb_i2c仅配置为I2C从机模式，此步骤可忽略。

1. 变量POLL_COUNT初始化为零。
2. 将IC_ENABLE寄存器的第0位清零。
3. 读取IC_ENABLE_STATUS寄存器，并检测IC_EN位（第0位）。将 POLL_COUNT 加一。若 POLL_COUNT >= MAX_T_POLL_COUNT，则返回相应错误代码并退出。
4. 若 IC_ENABLE_STATUS[0] 为 1，则休眠 t_{i2c_poll} 并返回至前一步。否则，返回相应成功代码并退出。

4.3.10.4. 中止 I2C 传输

IC_ENABLE 寄存器中的 ABORT 控制位允许软件在未完成从 Tx FIFO 发出的传输命令前释放 I2C 总线。接收到 ABORT 请求时，控制器会在 I2C 总线上发出 STOP 条件，随后执行 Tx FIFO 清空。仅允许运行于主机模式时中止传输。

4.3.10.4.1. 操作步骤

1. 停止向 Tx FIFO (IC_DATA_CMD) 写入新命令。
2. 在 DMA 模式下，将 TDMAE 清零以禁用发送 DMA。
3. 将 IC_ENABLE.ABORT 置为 1。
4. 等待 M_TX_ABRT 中断信号。

5. 读取 IC_TX_ABRT_SOURCE 寄存器以识别源为 ABRT_USER_ABRT。

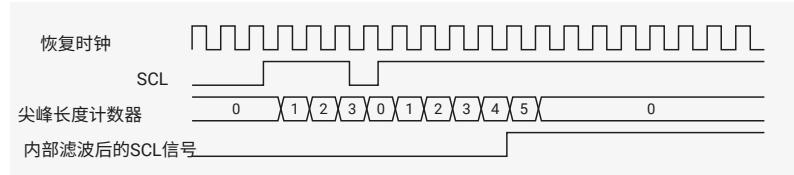
4.3.11. 尖峰抑制

DW_apb_i2c 包含可编程的脉冲抑制逻辑，以满足 I2C 总线规范中对 SS/FS 模式的要求。该逻辑基于计数器，监测输入信号（**SCL** 和 **SDA**），核查其在采样前是否在预定数量的 **ic_clk** 周期内保持稳定。每个信号（**SCL** 和 **SDA**）均配备独立的计数器。计数的 **ic_clk** 周期数由用户可编程，并应根据 **ic_clk** 频率及相关脉冲宽度规范计算确定。每当输入信号的值发生变化时，相应的计数器即被启动。依据输入信号的行为，将出现以下情形之一：

- 输入信号保持不变，直至计数器达到其计数上限值。此时，内部信号值更新为输入值，计数器复位并停止。计数器在检测到输入信号的新变化之前不会重新启动。
- 计数器达到计数限制值之前，输入信号已再次发生变化。出现此情况时，计数器被复位并停止，但信号的内部版本未更新。计数器将保持停止状态，直至检测到输入信号的新变化。

图83中的时序图说明了上述行为。

图83。尖峰抑制示例



① 注意

SCL 输入端设有两级同步器，但为简化起见，图83的时序图未包含该同步延迟。

I2C总线规范根据工作模式对最大尖峰长度有不同要求——SS和FS模式为50纳秒，因此该寄存器用于存储所需值：

- 寄存器IC_FS_SPKLEN保存SS和FS模式下的最大尖峰长度值

该寄存器位宽为8位，可通过APB接口进行读写访问；然而，仅当 DW_apb_i2c 被禁用时，才允许写入这些寄存器。这些寄存器可编程的最小值为 1；尝试编程小于 1 的值时，实际写入的值为 1。

这些寄存器的默认值基于 **ic_clk** 周期为 100ns，应根据 RP2040 上使用的 **clk_sys** 周期进行更新。

注意

- 由于 IC_FS_SPKLEN 寄存器的最小可编程值为 1，低频 ic_clk 下尖峰长度规格可能被超出。以 ic_clk 为 10MHz（周期为 100ns）为例；在此情况下，可编程的最小尖峰长度为 100ns，因此可抑制最长达此长度的尖峰。
- 标准同步逻辑（两个串联触发器）位于尖峰抑制逻辑的上游，其不受尖峰长度寄存器内容或尖峰抑制逻辑操作的任何影响；这两个操作（同步和脉冲抑制）完全独立。
由于 SCL 和 SDA 输入信号相对于 ic_clk 是异步的，对这些信号的采样存在一个 ic_clk 周期的不确定性；即根据它们相对于 ic_clk 上升沿发生的时间不同，采样后同一原始长度的脉冲可能出现一个 ic_clk 周期的差异。
- 脉冲抑制是对称的；即从零跳变到一和从一跳变到零的行为完全相同。

4.3.12. 快速模式 Plus 操作

在快速模式加（fast mode plus）下，DW_apb_i2c 允许将快速模式操作扩展至支持高达 1000kbps 的速率。要启用 DW_apb_i2c 的快速模式加操作，请在启动任何数据传输前完成以下步骤：

- 将 ic_clk 频率设置为大于或等于 32MHz（参见第 4.3.14.2.1 节）。
- 将 IC_CON 寄存器[2:1]编程为 2'b10，以支持快速模式或快速模式加。
- 编程 IC_FS_SCL_LCNT 和 IC_FS_SCL_HCNT 寄存器，以满足快速模式加 SCL 的要求（参见第 4.3.14 节）。
- 编程 IC_FS_SPKLEN 寄存器，以抑制最大 50 纳秒的尖峰脉冲。
- 编程 IC_SDA_SETUP 寄存器，以满足最小数据建立时间（tSU; DAT）要求。

4.3.13. 总线清除功能

DW_apb_i2c 支持总线清除功能，该功能能够在时钟线或数据线异常被拉低的少见情况下，优雅地恢复数据 SDA 和时钟 SCL 线。

4.3.13.1. SDA 线异常被拉低的恢复

若 SDA 线异常拉低，主设备将执行下述操作进行恢复，如图 84 和图 85 所示：

- 主设备发送最多九个时钟脉冲，以在该九个时钟周期内恢复总线低电平状态。
 - 时钟脉冲的数量将随从设备剩余待发送位数的变化而变化。由于最大位数为九位，主设备发送最多九个时钟脉冲，并允许从设备进行恢复。
 - 主设备尝试在 SDA 线上施加逻辑 1，并检测 SDA 是否被恢复。如果 SDA 未被恢复，主设备将继续发送最多九个 SCL 时钟脉冲。
- 若 SDA 线在九个时钟脉冲内恢复，主设备将发送停止信号以释放总线。
- 若在第九个时钟脉冲后 SDA 线仍未恢复，系统需执行硬件复位。

图84. SDA
使用9个 **SCL** 时钟进行恢复时钟

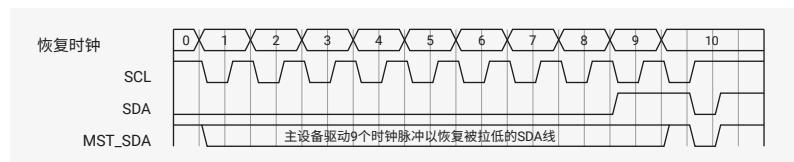
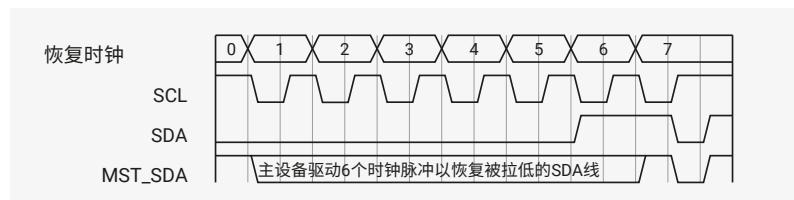


图85. SDA
使用6个 **SCL** 时钟进行恢复时钟



4.3.13.2. **SCL** 线被长时间拉低

在极少发生的情况下（由于电路电气故障），若时钟（**SCL**）保持低电平且无法恢复，则唯一有效的解决方法是通过硬件复位信号重置总线。

4.3.14. **IC_CLK** 频率配置

当DW_apb_i2c配置为标准模式（SS）、快速模式（FS）或快速增强模式（FM+）时，必须在任何I2C总线事务开始前设置*CNT寄存器，以确保正确的I/O时序。*CNT寄存器包括：

- [IC_SS_SCL_HCNT](#)
- [IC_SS_SCL_LCNT](#)
- [IC_FS_SCL_HCNT](#)
- [IC_FS_SCL_LCNT](#)

注意

tBUF时序及START、STOP和RESTART寄存器的建立/保持时间均采用对应速度模式的*HCNT/*LCNT寄存器设置。

注意

若DW_apb_i2c仅配置为I2C从设备运行，则无需配置任何*CNT寄存器，因这些寄存器仅用于确定I2C主设备模式下 **SCL** 的时序要求。

表449列出了基于*CNT编程寄存器推导I2C时序参数的方法。

表449。从*CNT寄存器推导I2C时序参数

时序参数	符号	标准速率	快速速率 / 快速速率增强
SCL 时钟的低电平周期	tLOW	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
SCL 时钟的高电平周期	tHIGH	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
重复启动条件的建立时间	tSU;STA	IC_SS_SCL_LCNT	IC_FS_SCL_HCNT
保持时间（重复）启动条件*	tHD;STA	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
停止条件的建立时间	tSU;STO	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT

时序参数	符号	标准速率	快速速率 / 快速速率增强
STOP与START条件之间的总线空闲时间	tBUF	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
脉冲宽度	tSP	IC_FS_SPKLEN	IC_FS_SPKLEN
数据保持时间	tHD;DAT	IC_SDA_HOLD	IC_SDA_HOLD
数据建立时间	tSU;DAT	IC_SDA_SETUP	IC_SDA_SETUP

4.3.14.1. SS、FS 及 FM+ 模式下的最小高低计数

当 DW_apb_i2c 作为 I2C 主设备运行时，在发送和接收传输过程中：

- IC_SS_SCL_LCNT 与 IC_FS_SCL_LCNT 寄存器值必须大于 IC_FS_SPKLEN + 7。
- IC_SS_SCL_HCNT 与 IC_FS_SCL_HCNT 寄存器值必须大于 IC_FS_SPKLEN + 5。

有关 DW_apb_i2c 高低计数的详细说明如下：

- *_LCNT 寄存器的最小值为 IC_*_SPKLEN + 7，该数值基于 DW_apb_i2c 在 SCL 下降沿后驱动 SDA 所需时间。
- *_HCNT 寄存器的最小值为 IC_*_SPKLEN + 5，该数值基于 DW_apb_i2c 在 SCL 高电平期间采样 SDA 所需时间。
- DW_apb_i2c 在编程的 *_LCNT 值基础上增加一个周期，以产生 SCL 时钟的低电平周期；这是因为 SCL 低电平计数逻辑计数至 (*_LCNT + 1)。
- DW_apb_i2c 在编程的 *_HCNT 值基础上增加 IC_*_SPKLEN + 7 个周期，以产生 SCL 时钟的高电平周期；其原因如下：
 - SCL 高电平计数逻辑计数至 (*_HCNT + 1)。
 - 施加于 SCL 线的数字滤波导致延迟为 SPKLEN + 2 个 ic_clk 周期，其中 SPKLEN 定义为：
 - 当组件工作于 SS 或 FS 模式时，为 IC_FS_SPKLEN
 - 每当 DW_apb_i2c 将 SCL 信号由高拉低——即完成 SCL 的高电平周期时——内部逻辑会产生三个 ic_clk 周期的延迟。因此，DW_apb_i2c 所能支持的最小 SCL 低电平时间为九个 ic_clk 周期 (7 + 1 + 1)，而最小 SCL 高电平时间为十三个 ic_clk 周期 (6 + 1 + 3) + 3)。

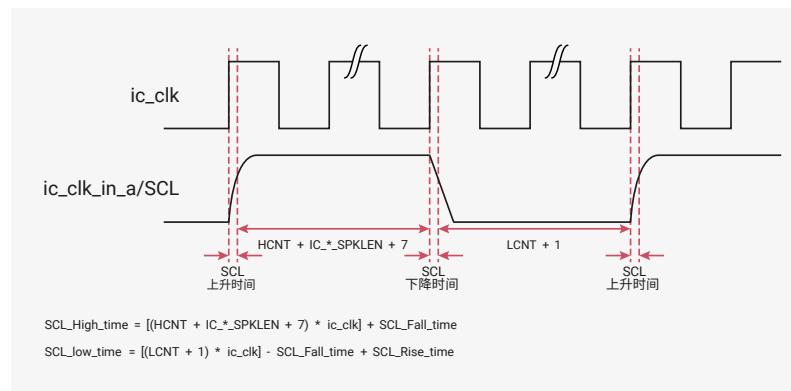
i 注意

DW_apb_i2c 主设备产生的 SCL 总高电平时间和低电平时间还受到 SCL 线上升时间和下降时间的影响，如图 86 中的示意图及方程所示。需注意，SCL 上升和下降时间参数因外部因素不同而变化，例如：

- IO 驱动器特性
- 上拉电阻值
- SCL 线上总电容等

上述特性均超出 DW_apb_i2c 的控制范围。

图86。 **SCL**上升时间与下降时间对生成信号的影响



4.3.14.2. 最小 **IC_CLK** 频率

本节说明了 DW_apb_i2c 针对各速率模式支持的最小 **ic_clk** 频率及其对应的高低计数值。在从设备模式下，需对 IC_SDA_HOLD (Thd:dat) 和 IC_SDA_SETUP (Tsu:dat) 进行编程，以满足 I2C 协议的时序要求。以下示例适用于 IC_FS_SPKLEN 被设定为 2 的情况。

4.3.14.2.1. 标准模式 (SM)、快速模式 (FM) 及快速模式增强型 (FM+)

本节详细说明如何为 DW_apb_i2c 的标准模式和快速模式导出最小 **ic_clk** 值。尽管以下方法展示了快速模式的计算过程，但同样适用于标准模式及快速加模式的计算。

注意

以下计算未考虑 SCL_Rise_time 和 SCL_Fall_time。

快速模式下 DW_apb_i2c 最小 **ic_clk** 值的条件与计算如下：

- 快速模式的数据传输速率为 400 kbps；即 **SCL** 周期为 $1/400\text{kHz} = 2.5\mu\text{s}$
- 最小初始 hcnt 值为 14； $IC_HCNT_FS = 14$
- 协议规定的最小 **SCL** 高电平时间与低电平时间如下：
 - $MIN_SCL_LOWtime_FS = 1300\text{ns}$
 - $MIN_SCL_HIGHtime_FS = 600\text{ns}$

推导公式：

$$SCL_PERIOD_FS / (IC_HCNT_FS + IC_LCNT_FS) = IC_CLK_PERIOD$$

$$IC_LCNT_FS \times IC_CLK_PERIOD = MIN_SCL_LOWtime_FS$$

综合前述方程，得到如下结果：

$$IC_LCNT_FS \times (SCL_PERIOD_FS / (IC_LCNT_FS + IC_HCNT_FS)) = MIN_SCL_LOWtime_FS$$

求解 IC_LCNT_FS 如下：

$$\text{IC_LCNT_FS} \times (2.5\mu\text{s} / (\text{IC_LCNT_FS} + 14)) = 1.3 \mu\text{s}$$

上述方程给出：

$$\text{IC_LCNT_FS} = \text{roundup}(15.166) = 16$$

计算结果为 $\text{IC_LCNT_FS} = 16$, $\text{IC_HCNT_FS} = 14$, 对应的 ic_clk 值为：

$$2.5\mu\text{s} / (16 + 14) = 83.3\text{ns} = 12\text{MHz}$$

经验证，该结果满足协议要求。

表450列出了所有模式下高计数与低计数对应的最小 ic_clk 值。

表450. ic_clk 与高计数与低计数的关系

速度模式	ic_clkfreq (MHz)	IC_SPKLEN 的最小值	SCL 低电平时间, 以 ic_clk 为单位	SCL 低电平程序设定值	SCL 低电平时间	SCL 高电平时间, 单位为 ic_clk	SCL 高电平程序设定值	SCL 高电平时间
SS	2.7	1	13	12	$4.7\mu\text{s}$	14	6	$5.2\mu\text{s}$
FS	12.0	1	16	15	$1.33\mu\text{s}$	14	6	$1.16\mu\text{s}$
FM+	32	2	16	15	500ns	16	7	500ns

- IC_*_SCL_LCNT 和 IC_*_SCL_HCNT 寄存器通过表450中的 SCL 低电平和高电平程序设定值编程, 该值分别由 SCL 低电平时间减一及 SCL 高电平时间减八计算得出。表450中的数值基于 $\text{IC_SDA_RX_HOLD} = 0$ 。最大 IC_SDA_RX_HOLD 值取决于主模式下的 IC_*CNT 寄存器。

- 为计算考虑RC时序的HCNT和LCNT, 使用以下公式:

- $\text{IC_HCNT_*} = [(\text{HCNT} + \text{IC_SPKLEN} + 7) * \text{ic_clk}] + \text{SCL_Fall_time}$
- $\text{IC_LCNT_*} = [(\text{LCNT} + 1) * \text{ic_clk}] - \text{SCL_Fall_time} + \text{SCL_Rise_time}$

4.3.14.3 高电平和低电平计数的计算

以下计算说明如何为 DW_apb_i2c 中各速率模式计算 SCL 高电平和低电平计数。为确保计算有效, 所用 ic_clk 频率不得低于表450中规定的最低 ic_clk 频率。

默认 ic_clk 周期值为 100ns, 基于此时钟计算各速率模式的默认 SCL 高电平和低电平计数值。这些数值需依据以下指南进行更新。

用于计算设置合适 SCL 时钟高、低电平时间所需 ic_clk 信号数量的公式如下：

$$\text{IC_XCNT} = (\text{ROUNDUP}(\text{MIN_SCL_xxxtime} * \text{OSCFREQ}, 0))$$

MIN_SCL_HIGHtime = 最小高电平周期

MIN_SCL_HIGHtime = 100kbps 时为 4000ns, 400
kbps 时为 600ns, 100
0kbps 时为 260ns。

MIN_SCL_LOWtime = 最小低电平周期

MIN_SCL_LOWtime = 100kbps 时为 4700ns,

400kbps时为1300ns,
1000kbps时为500ns。

$\text{OSCFREQ} = \text{ic_clk}$ 时钟频率 (Hz)

例如：

```

OSCFREQ = 100MHz
I2Cmode = 快速模式, 400kbps
MIN_SCL_HIGHtime = 600ns。
MIN_SCL_LOWtime = 1300ns。

IC_xCNT = (ROUNDUP(MIN_SCL_HIGH_LOWtime*OSCFREQ, 0))

IC_HCNT = (ROUNDUP(600ns * 100MHz, 0))
IC_HCNTSCL 周期 = 60
IC_LCNT = (ROUNDUP(1300ns * 100MHz, 0))
IC_LCNTSCL 周期 = 130
实际最小 SCL 高电平时间 = 60 × (1/100MHz) = 600ns
实际最小 SCL 低电平时间 = 130 × (1/100MHz) = 1300ns

```

4.3.15. DMA 控制器接口

DW_apb_i2c 具备内置 DMA 功能；其通过握手接口与 DMA 控制器通信，以请求并控制传输。APB 总线用于在 DMA 之间执行数据传输。由于数据速率较低，DMA 传输以单次访问的形式进行。

4.3.15.1. 启用 DMA 控制器接口

要启用 DW_apb_i2c 上的 DMA 控制器接口，必须写入 DMA 控制寄存器 (IC_DMA_CR)。
向 IC_DMA_CR 寄存器的 TDMAE 位域写入 1 以启用 DW_apb_i2c 的发送握手机制。
向 IC_DMA_CR 寄存器的 RDMAE 位域写入 1 以启用 DW_apb_i2c 的接收握手机制。

4.3.15.2. 操作概述

DMA 控制器被编程为传输或接收 DW_apb_i2c 指定数量的数据项（传输计数）。

传输被拆分为总线上的单个传输，每次由 DW_apb_i2c 发起请求。

例如，若 DMA 控制器中设置的传输计数为四。DMA 传输包含连续的四次单独事务。若 DW_apb_i2c 向该通道发起发送请求，则会向 DW_apb_i2c 的 TX FIFO 写入一个数据项。同理，若 DW_apb_i2c 向该通道发起接收请求，则会从 DW_apb_i2c 的 RX FIFO 读取一个数据项。完成所有四个数据项的读写之前，必须对该 DMA 通道发起四次独立请求。

4.3.15.3. 水印级别

在 DW_apb_i2c 中，用于设置水印以允许 DMA 突发的寄存器无需设置为除复位值外的其他值。具体而言，IC_DMA_TDLR 和 IC_DMA_RDLR 可保持为复位值零。这是因为相较于系统带宽，I2C 的带宽较低，仅需单次传输，同时 DMA 控制器

通常在系统总线上优先级最高，因此一般能非常快速地完成传输。

4.3.16. 中断寄存器操作

表451列出了 DW_apb_i2c 中断寄存器的操作方式及其设置和清除的方法。某些位由硬件设置并由软件清除，而其他位则由硬件设置和清除。

表451。中断寄存器的清除与设置

中断位字段	硬件设置／软件清除	硬件设置与清除
RESTART_DET	是	否
GEN_CALL	是	否
START_DET	是	否
STOP_DET	是	否
活动	是	否
RX_DONE	是	否
TX_ABRT	是	否
RD_REQ	是	否
TX_EMPTY	否	是
TX_OVER	是	否
RX_FULL	否	是
RX_OVER	是	否
RX_UNDER	是	否

4.3.17. 寄存器列表

I2C0 和 I2C1 寄存器的基址分别为 `0x40044000` 和 `0x40048000`（在 SDK 中定义为 I2C0_BASE 和 I2C1_BASE）。

① 注意

您可能会在I2C寄存器描述中看到配置常量的引用；这些值是固定的，于硬件设计阶段设定。完整的取值列表可见于 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_regs/include/hardware/regs/i2c.h

表452. I2C 寄存器列表

偏移量	名称	说明
0x00	IC_CON	I2C控制寄存器
0x04	IC_TAR	I2C目标地址寄存器
0x08	IC_SAR	I2C从属地址寄存器
0x10	IC_DATA_CMD	I2C接收/发送数据缓冲及命令寄存器
0x14	IC_SS_SCL_HCNT	标准速度I2C时钟SCL高电平计数寄存器
0x18	IC_SS_SCL_LCNT	标准速度I2C时钟SCL低电平计数寄存器
0x1c	IC_FS_SCL_HCNT	快速模式或增强快速模式I2C时钟SCL高电平计数寄存器
0x20	IC_FS_SCL_LCNT	快速模式或增强快速模式I2C时钟SCL低电平计数寄存器

偏移量	名称	说明
0x2c	IC_INTR_STAT	I2C中断状态寄存器
0x30	IC_INTR_MASK	I2C中断屏蔽寄存器
0x34	IC_RAW_INTR_STAT	I2C原始中断状态寄存器
0x38	IC_RX_TL	I2C接收FIFO阈值寄存器
0x3c	IC_TX_TL	I2C发送FIFO阈值寄存器
0x40	IC_CLR_INTR	清除组合中断及单个中断寄存器
0x44	IC_CLR_RX_UNDER	清除 RX_UNDER 中断寄存器
0x48	IC_CLR_RX_OVER	清除 RX_OVER 中断寄存器
0x4c	IC_CLR_TX_OVER	清除 TX_OVER 中断寄存器
0x50	IC_CLR_RD_REQ	清除 RD_REQ 中断寄存器
0x54	IC_CLR_TX_ABRT	清除 TX_ABRT 中断寄存器
0x58	IC_CLR_RX_DONE	清除 RX_DONE 中断寄存器
0x5c	IC_CLR_ACTIVITY	清除 ACTIVITY 中断寄存器
0x60	IC_CLR_STOP_DET	清除 STOP_DET 中断寄存器
0x64	IC_CLR_START_DET	清除 START_DET 中断寄存器
0x68	IC_CLR_GEN_CALL	清除 GEN_CALL 中断寄存器
0x6c	IC_ENABLE	I2C 使能寄存器
0x70	IC_STATUS	I2C 状态寄存器
0x74	IC_TXFLR	I2C 发送 FIFO 水平寄存器
0x78	IC_RXFLR	I2C 接收 FIFO 水平寄存器
0x7c	IC_SDA_HOLD	I2C SDA 保持时间寄存器
0x80	IC_TX_ABRT_SOURCE	I2C 发送中止源寄存器
0x84	IC_SLV_DATA_NACK_ONLY	生成从设备数据 NACK 寄存器
0x88	IC_DMA_CR	DMA 控制寄存器
0x8c	IC_DMA_TDLR	DMA 传输数据级别寄存器
0x90	IC_DMA_RDLR	DMA 传输数据级别寄存器
0x94	IC_SDA_SETUP	I2C SDA 设置寄存器
0x98	IC_ACK_GENERAL_CALL	I2C 应答通用调用寄存器
0x9c	IC_ENABLE_STATUS	I2C 使能状态寄存器
0xa0	IC_FS_SPKLEN	I2C SS、FS 或 FM+ 峰值抑制限制
0xa8	IC_CLR_RESTART_DET	清除 RESTART_DET 中断寄存器
0xf4	IC_COMP_PARAM_1	组件参数寄存器 1
0xf8	IC_COMP_VERSION	I2C 组件版本寄存器
0xfc	IC_COMP_TYPE	I2C 组件类型寄存器

I2C: IC_CON 寄存器

偏移: 0x00

描述

I2C 控制寄存器。该寄存器仅在 DW_apb_i2c 被禁用时可写，对应 IC_ENABLE[0] 寄存器设为 0。其他时间的写入操作无效。

读/写权限：- 位 10 为只读。- 位 11 为只读 - 位 16 为只读 - 位 17 为只读 - 位 18 和位 19 为只读。

表 453. IC_CON
寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	STOP_DET_IF_MASTER_ACTIVE : 无论主设备是否处于激活状态，主机均会触发 STOP_DET 中断	只读	0x0
9	RX_FIFO_FULL_HLD_CTRL : 该位控制当接收 FIFO 物理填满至 RX_BUFFER_D_EPTH 时，DW_apb_i2c 是否应阻止总线访问，如 IC_RX_FULL_HLD_BU_S_EN 参数中所述。 复位值: 0x0。	读写	0x0
	枚举值:		
	0x0 → 禁用: 当 RX_FIFO 满时发生溢出		
	0x1 → 启用: 当 RX_FIFO 满时保持总线		
8	TX_EMPTY_CTRL : 该位控制 TX_EMPTY 中断的生成，如 IC_RAW_INTR_STA_T 寄存器所述。 复位值: 0x0。	读写	0x0
	枚举值:		
	0x0 → 禁用: TX_EMPTY 中断的默认行为		
	0x1 → 启用: 控制 TX_EMPTY 中断的生成		
7	STOP_DET_IFADDRESSED : 在从属模式下: - 1'b1: 仅当被寻址时发出 STOP_DET 中断。- 1'b0: 无论是否被寻址，均发出 STOP_DET 中断。复位值: 0x0 注意: 在通用调用地址期间，如 STOP_DET_IF_ADDRESSED=1'b1，即使从属通过生成 ACK 对通用调用地址作出响应，亦不会发出 STOP_DET 中断。STOP_DET 中断仅在所传输地址与从属地址 (SAR) 匹配时生成。	读写	0x0
	枚举值:		
	0x0 → 已禁用: 从机始终发出 STOP_DET 中断		
	0x1 → 已启用: 从机仅在被寻址时发出 STOP_DET 中断		
6	IC_SLAVE_DISABLE : 此位控制 I2C 从机是否被禁用，即一旦施加预设信号，该位被置位，从机即被禁用。 如果该位被置位（从机被禁用），则 DW_apb_i2c 仅作为主机工作，不执行任何需要从机的操作。 注意: 软件应确保如果该位写入 0，则位0也应写入 0。	读写	0x1

位	描述	类型	复位值
	枚举值：		
	0x0 → 从机已启用：从机模式被启用		
	0x1 → 从机已禁用：从机模式被禁用		
5	IC_RESTART_EN : 决定作为主机时是否允许发送 RESTART 条件。部分旧版从机不支持处理 RESTART 条件；然而，在多个 DW_apb_i2c 操作中会使用 RESTART 条件。当禁用 RESTART 时，主设备禁止执行以下操作：- 发送 START 字节 - 执行任何高速模式操作 - 在组合格式模式下进行方向切换 - 使用 10 位地址执行读取操作。通过用 RESTART 条件替代随后紧跟的 STOP 以及再次的 START 条件，分割操作被拆分为多个 DW_apb_i2c 传输。若执行上述操作，将导致设置 IC_RAW_INTR_STAT 寄存器第 6 位 (TX_AB RT)。	读写	0x1
	复位值：ENABLED		
	枚举值：		
	0x0 → DISABLED：主设备重启功能被禁用		
	0x1 → ENABLED：主设备重启功能被启用		
4	IC_10BITADDR_MASTER : 控制 DW_apb_i2c 作为主设备时，传输启动所采用的 7 位或 10 位寻址模式。- 0: 7 位寻址 - 1: 10 位寻址	读写	0x0
	枚举值：		
	0x0 → ADDR_7BITS：主设备 7 位寻址模式		
	0x1 → ADDR_10BITS：主机10位寻址模式		
3	IC_10BITADDR_SLAVE : 作为从机时，该位控制DW_apb_i2c响应7位还是10位地址。- 0: 7位寻址。DW_apb_i2c会忽略涉及10位寻址的事务；对于7位寻址，仅比较IC_SAR寄存器的低7位。- 1: 10位寻址。DW_apb_i2c仅响应与IC_SAR寄存器完整10位匹配的10位寻址传输。	读写	0x0
	枚举值：		
	0x0 → ADDR_7BITS：从机7位寻址		
	0x1 → ADDR_10BITS：从机10位寻址		

位	描述	类型	复位值
2:1	<p>SPEED: 这些位用于控制DW_apb_i2c的工作速度；该设置仅在以主机模式操作DW_apb_i2c时适用。 硬件防止非法值被软件编入。 这些位也必须为从模式适当编程，因为它们用于根据速度模式捕获峰值滤波器的正确值。</p> <p>此寄存器应仅编程为1至IC_MAX_SPEED_MODE范围内的值；否则，硬件会将此寄存器更新为IC_MAX_SPEED_MODE的值。</p> <p>1: 标准模式 (100 kbit/s) 2: 快速模式 (<=400 kbit/s) 或快速增强模式 (<=1000 kbit/s) 3: 高速模式 (3.4 Mbit/s)</p> <p>注：当IC_ULTRA_FAST_MODE=1时，此字段不适用。</p>	读写	0x2
	枚举值：		
	0x1 → STANDARD：标准速度操作模式		
	0x2 → FAST：快速或快速增强操作模式		
	0x3 → HIGH：高速操作模式		
0	<p>MASTER_MODE: 该位控制是否启用DW_apb_i2c主控。</p> <p>注意：软件应确保若此位写入‘1’，则第6位亦应写入‘1’。</p>	读写	0x1
	枚举值：		
	0x0 → 禁用：主模式被禁用		
	0x1 → 启用：主模式被启用		

I2C: IC_TAR寄存器

偏移量: 0x04

说明

I2C目标地址寄存器

该寄存器宽度为12位，第31至12位为保留位。仅当 IC_ENABLE[0] 设置为0时，方可写入该寄存器。

注意：若软件或应用程序确认 DW_apb_i2c 未将 TAR 地址用于 Tx FIFO 中的待处理命令，则即使 Tx FIFO 有条目 (IC_STATUS[2]=0)，仍可更新 TAR 地址。- 若 DW_apb_i2c 仅作为 I2C 从设备启用，则无需对该寄存器进行写入操作。

表454. IC_TAR
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	<p>SPECIAL: 该位指示软件是否执行设备ID、通用呼叫或起始字节命令。- 0: 忽略第10位GC_OR_START，正常使用IC_TAR - 1: 执行Device_ID或GC_OR_START位指定的特殊I2C命令 复位值: 0x0</p>	读写	0x0

位	描述	类型	复位值
	枚举值：		
	0x0 → 禁用：禁止编程GENERAL_CALL或START_BYT传输		
	0x1 → 启用：允许编程GENERAL_CALL或START_BYT传输		
10	GC_OR_START : 当第11位 (SPECIAL) 为1且第13位 (Device-ID) 为0时，此位指示DW_apb_i2c是否执行General Call或START字节命令。- 0: General Call 地址 - 发出General Call后，仅允许写操作。尝试发出读取命令时，将设置IC_RAW_INTR_STAT寄存器的第6位 (TX_ABRT)。DW_apb_i2c将保持在General Call模式，直至SPECIAL位 (第11位) 被清除。- 1: START BYTE 重置值: 0x0	读写	0x0
	枚举值：		
	0x0 → GENERAL_CALL：GENERAL_CALL 字节传输		
	0x1 → START_BYT：START 字节传输		
9:0	IC_TAR : 此地址为任何主设备事务的目标地址。发送 General Call 时，这些位将被忽略。生成 START BYT 时，CPU 只需写入这些位一次即可。 若 IC_TAR 与 IC_SAR 相同，则存在回环，但主从共享 FIFO，故不支持完全回环。仅支持单向回环模式（单工），不支持双工模式。主设备不可向自身发送数据；只能向从设备发送数据。	读写	0x055

I2C: IC_SAR 寄存器

偏移: 0x08

描述

I2C从属地址寄存器

表 455. IC_SAR 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:0	IC_SAR : I2C 作为从设备时，IC_SAR 保存从设备地址。对于7位寻址，仅使用IC_SAR[6:0]。 仅当I2C接口被禁用（对应IC_ENABLE[0]寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。 注意：默认值不得为任何保留地址区间内的地址，即0x00至0x07，或0x78至0x7F。若将IC_SAR或IC_TAR设置为保留值，设备的正确运行将无法保证。 有关完整保留值列表，请参见表448。	读写	0x055

I2C: IC_DATA_CMD寄存器

偏移: 0x10

描述

I2C接收/发送数据缓冲及命令寄存器；CPU向TX FIFO写入数据及从RX FIFO读取字节时，均通过此寄存器进行操作。

寄存器位宽变化如下：

写入： - 当IC_EMPTYFIFO_HOLD_MASTER_EN=1时为11位 - 当IC_EMPTYFIFO_HOLD_MASTER_EN=0时为9位
 读取： - 当IC_FIRST_DATA_BYTE_STATUS=1时为12位 - 当IC_FIRST_DATA_BYTE_STATUS=0时为8位
 注：为确保DW_apb_i2c持续响应读取，每个待接收字节应对应写入一个读取命令；
 否则 DW_apb_i2c 将停止响应确认。

表 456。
IC_DATA_CMD
 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	FIRST_DATA_BYTE : 表示在主设备接收或从设备接收模式下，地址阶段后接收到的第一个数据字节。 复位值：0x0 注意：当 APB_DATA_WIDTH=8 时， 1. 用户必须对 IC_DATA_CMD 执行两次 APB 读取，以获取第 11 位的状态。 2. 为读取第 11 位，用户需首先读取第一个数据字节 [7:0]（偏移地址 0x10），随后执行第二次读取 [15:8]（偏移地址 0x11），以确定第 11 位状态（即上次读取的数据是否为第一个数据字节）。 3. 第 11 位为可选读取字段，用户可忽略第二次读取的 [15:8]（偏移地址 0x11），若不关心 FIRST_DATA_BYTE 状态。	只读	0x0
	枚举值：		
	0x0 → 非活动：接收到顺序数据字节		
	0x1 → 激活：接收到非顺序数据字节		
10	RESTART : 该位控制是否在数据字节发送或接收之前发出 RESTART。 1 - 若 IC_RESTART_EN 为 1，则无论传输方向是否改变，均会在数据发送/接收之前（依据 CMD 值）发出 RESTART；若 IC_RESTART_EN 为 0，则改为先发出 STOP 再发出 START。 0 - 若 IC_RESTART_EN 为 1，仅当传输方向与前一命令不同，才发出 RESTART；若 IC_RESTART_EN 为 0，则改为先发出 STOP 再发出 START。 复位值：0x0	SC	0x0
	枚举值：		
	0x0 → 禁用：此命令前不发出 RESTART		
	0x1 → 启用：此命令前发出 RESTART		

位	描述	类型	复位值
9	<p>STOP: 该位控制是否在数据字节发送或接收之后发出 STOP。</p> <p>- 1 - 无论 Tx FIFO 是否为空，此字节后均发出 STOP 信号。如果 Tx FIFO 非空，主设备将立即尝试通过发出 START 信号并竞争总线来启动新的传输。- 0 - 无论 Tx FIFO 是否为空，此字节后均不发出 STOP 信号。如果 Tx FIFO 非空，主设备根据 CMD 位的值继续通过发送/接收数据字节完成当前传输。如果 Tx FIFO 为空，主设备将保持 SCL 线为低电平，暂停总线，直到 Tx FIFO 中有新的命令。复位值：0x0</p>	SC	0x0
	枚举值：		
	0x0 → 禁用：此命令后不发出 STOP 信号		
	0x1 → 启用：此命令后发出 STOP 信号		
8	<p>CMD: 此位控制执行读操作或写操作。当 DW_apb_i2con 作为从设备时，此位不控制方向。仅在作为主设备时控制方向。</p> <p>当命令输入 TX FIFO 时，此位用于区分写命令和读命令。在从设备接收模式下，此位为“无关位”，因无需写入该寄存器。在从设备发送模式下，‘0’表示将传输 IC_DATA_CMD 中的数据。</p> <p>编程设置此位时，请注意：若在发送 General Call 命令后尝试执行读操作，除非已清除 IC_TAR 寄存器第 11 位（SPECIAL），否则会触发 TX_ABRT 中断（IC_RA_W_INTR_STAT 寄存器第 6 位）。若在接收到 RD_REQ 中断后向此位写入 ‘1’，则会引发 TX_ABRT 中断。</p> <p>复位值：0x0</p>	SC	0x0
	枚举值：		
	0x0 → WRITE：主设备写命令		
	0x1 → READ：主设备读命令		
7:0	<p>DAT: 该寄存器包含通过 I2C 总线传输或接收的数据。写入该寄存器并意图进行读操作时，DW_apb_i2c 会忽略位 7:0 (DAT)；读取该寄存器时，这些位返回 DW_apb_i2c 接口接收的数据值。</p> <p>复位值：0x0</p>	读写	0x00

I2C: IC_SS_SCL_HCNT 寄存器

偏移量: 0x14

描述

标准速度 I2C 时钟 SCL 高电平计数寄存器

表457。
IC_SS_SCL_HCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-

位	描述	类型	复位值
15:0	<p>IC_SS_SCL_HCNT: 必须在任何 I2C 总线事务开始之前设置此寄存器，以确保正确的输入/输出时序。该寄存器设置标准速率下 SCL 时钟的高电平周期计数。更多信息请参阅“IC_CLK 频率配置”。</p> <p>仅当 I2C 接口被禁用时（对应 IC_ENABLE[0] 寄存器设为 0），才允许写入此寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 6；硬件会禁止写入小于此值的数值，若尝试写入，将自动设置为 6。对于 APB_DATA_WIDTH = 8 的设计，编程顺序至关重要，以确保 DW_apb_i2c 的正确运行。必须先编程低字节。然后编程高字节。</p> <p>注意：该寄存器的设置值不得高于 65525，因为 DW_apb_i2c 使用 16 位计数器，当计数器达到 IC_SS_SCL_HCNT + 10 时，表示 I2C 总线处于空闲状态。</p>	读写	0x0028

I2C: IC_SS_SCL_LCNT 寄存器

偏移：0x18

说明

标准速度I2C时钟SCL低电平计数寄存器

表 458。
IC_SS_SCL_LCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	<p>IC_SS_SCL_LCNT: 此寄存器必须在任何 I2C 总线事务开始前设置，以确保正确的 I/O 时序。此寄存器设定标准速率下 SCL 时钟的低电平周期计数。详情请参阅“IC_CLK 频率配置”。</p> <p>仅当 I2C 接口被禁用时（对应 IC_ENABLE[0] 寄存器设为 0），才允许写入此寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 8；硬件禁止写入小于此值的数，若尝试写入，将自动设为 8。对于 APB_DATA_WIDTH = 8 的设计，编程顺序至关重要，以保证 DW_apb_i2c 的正确操作。必须先编程低字节，然后编程高字节。</p>	读写	0x002f

I2C: IC_FS_SCL_HCNT 寄存器

偏移：0x1c

说明

快速模式或增强快速模式I2C时钟SCL高电平计数寄存器

表 459。
IC_FS_SCL_HCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-

位	描述	类型	复位值
15:0	<p>IC_FS_SCL_HCNT: 此寄存器必须在任何 I2C 总线事务开始前设置，以确保正确的 I/O 时序。此寄存器用于设置快速模式或快速模式增强的 SCL 时钟高电平周期计数。该寄存器应用于高速模式，用以发送主控代码及起始字节或通用呼叫。详情请参阅“IC_CLK 频率配置”。</p> <p>当 IC_MAX_SPEED_MODE 设为 standard 时，该寄存器将消失并变为只读，返回值恒为 0。仅当 I2C 接口被禁用（对应 IC_ENABLE[0] 寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 6；硬件禁止写入低于此值的数值，尝试写入时将自动设定为 6。对于 APB_DATA_WIDTH 为 8 的设计，编程顺序尤为关键，以确保 DW_apb_i2c 的正常运作，必须先写入低字节。然后编程高字节。</p>	读写	0x0006

I2C: IC_FS_SCL_LCNT 寄存器

偏移: 0x20

说明

快速模式或增强快速模式 I2C 时钟 SCL 低电平计数寄存器

表460。
IC_FS_SCL_LCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	<p>IC_FS_SCL_LCNT: 在执行任何 I2C 总线传输之前，必须先设置此寄存器以保证正确的输入输出时序。此寄存器用于设置快速模式下 SCL 时钟低电平周期计数。该寄存器应用于高速模式，用以发送主控代码及起始字节或通用呼叫。详情请参阅“IC_CLK 频率配置”。</p> <p>当 IC_MAX_SPEED_MODE = standard 时，该寄存器将失效并变为只读，返回值为 0。</p> <p>仅当 I2C 接口被禁用（对应 IC_ENABLE[0] 寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 8；硬件禁止写入小于此值的数值，若尝试写入，则会被设置为 8。对于 APB_DATA_WIDTH = 8 的设计，编程顺序极为重要，以确保 DW_apb_i2c 的正确操作。必须先编程低字节。然后编程高字节。若该值小于 8，则计数值会被更改为 8。</p>	读写	0x000d

I2C: IC_INTR_STAT 寄存器

偏移量: 0x2c

描述

I2C 中断状态寄存器

该寄存器的每一位都对应于 IC_INTR_MASK 寄存器中的掩码位。通过读取相应的中断清除寄存器，可清除这些位。这些位的未屏蔽原始版本可在 IC_RAW_INTR_STAT 寄存器中获取。

寄存器的内容。

表 461。
IC_INTR_STAT
寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	R_RESTART_DET: 关于 R_RESTART_DET 位的详细描述, 请参见 IC_RAW_INT R_STAT 寄存器。 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_RESTART_DET 中断处于非激活状态		
	0x1 → 激活: R_RESTART_DET 中断处于激活状态		
11	R_GEN_CALL: 详见 IC_RAW_INTR_STAT 中 R_GEN_CALL 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_GEN_CALL 中断处于非激活状态		
	0x1 → 激活: R_GEN_CALL 中断处于激活状态		
10	R_START_DET: 详见 IC_RAW_INTR_STAT 中 R_START_DET 位的描述。 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_START_DET 中断处于非激活状态		
	0x1 → 激活: R_START_DET 中断处于激活状态		
9	R_STOP_DET: 详见 IC_RAW_INTR_STAT 中 R_STOP_DET 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_STOP_DET 中断处于非激活状态		
	0x1 → 激活: R_STOP_DET 中断处于激活状态		
8	R_ACTIVITY: 详见 IC_RAW_INTR_STAT 中 R_ACTIVITY 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_ACTIVITY 中断处于非激活状态		
	0x1 → 活动: R_ACTIVITY 中断处于活动状态		
7	R_RX_DONE: 详见 IC_RAW_INTR_STAT 中 R_RX_DONE 位的描述。 复位值: 0x0	只读	0x0
	枚举值:		

位	描述	类型	复位值
	0x0 → 非活动：R_RX_DONE中断处于非活动状态		
	0x1 → 活动：R_RX_DONE中断处于活动状态		
6	R_TX_ABRT : 详见IC_RAW_INTR_STAT中R_TX_ABRT位的描述。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_ABRT中断处于非活动状态		
	0x1 → 活动：R_TX_ABRT中断处于活动状态		
5	R_RD_REQ : 详见IC_RAW_INTR_STAT中R_RD_REQ位的描述。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_RD_REQ中断处于非活动状态		
	0x1 → 活动：R_RD_REQ中断处于活动状态		
4	R_TX_EMPTY : 详见IC_RAW_INTR_STAT中R_TX_EMPTY位的描述。 。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_EMPTY中断处于非活动状态		
	0x1 → 活动：R_TX_EMPTY中断处于活动状态		
3	R_TX_OVER : 有关R_TX_OVER位的详细说明，请参见IC_RAW_INTR_STAT。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_OVER中断处于非活动状态		
	0x1 → 活动：R_TX_OVER中断处于活动状态		
2	R_RX_FULL : 有关R_RX_FULL位的详细说明，请参见IC_RAW_INTR_STAT。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_RX_FULL中断处于非活动状态		
	0x1 → 活动：R_RX_FULL中断处于活动状态		
1	R_RX_OVER : 有关R_RX_OVER位的详细说明，请参见IC_RAW_INT_R_STAT。 复位值：0x0	只读	0x0
	枚举值：		

位	描述	类型	复位值
	0x0 → 非活动：R_RX_OVER 中断处于非活动状态		
	0x1 → 活动：R_RX_OVER 中断处于活动状态		
0	R_RX_UNDER : 有关 R_RX_UNDER 位的详细说明，请参见 IC_RAW_INTR_STAT。复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：RX_UNDER 中断处于非活动状态		
	0x1 → 活动：RX_UNDER 中断处于活动状态		

I2C: IC_INTR_MASK 寄存器

偏移：0x30

描述

I2C 中断屏蔽寄存器。

这些位用于屏蔽其对应的中断状态位。该寄存器为低电平有效；值为0时屏蔽中断，值为1时取消屏蔽中断。

表462。
IC_INTR_MASK
寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	M_RESTART_DET : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_RESTART_DET 中断。 IC_INTR_STAT 寄存器。 复位值：0x0	读写	0x0
	枚举值：		
	0x0 → 启用：RESTART_DET 中断被屏蔽		
	0x1 → 禁用：RESTART_DET 中断未被屏蔽		
11	M_GEN_CALL : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_GEN_CALL 中断。 复位值：0x1	读写	0x1
	枚举值：		
	0x0 → 启用：GEN_CALL 中断被屏蔽		
	0x1 → 禁用：GEN_CALL 中断未被屏蔽		
10	M_START_DET : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_START_DET 中断。 复位值：0x0	读写	0x0
	枚举值：		
	0x0 → 启用：START_DET 中断被屏蔽		
	0x1 → 禁用：START_DET 中断未被屏蔽		

位	描述	类型	复位值
9	M_STOP_DET: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_STOP_DET 中断。 复位值: 0x0	读写	0x0
	枚举值:		
	0x0 → 已启用: STOP_DET 中断被屏蔽		
	0x1 → 已禁用: STOP_DET 中断未被屏蔽		
8	M_ACTIVITY: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_ACTIVITY 中断。 复位值: 0x0	读写	0x0
	枚举值:		
	0x0 → 已启用: ACTIVITY 中断被屏蔽		
	0x1 → 已禁用: ACTIVITY 中断未被屏蔽		
7	M_RX_DONE: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_DONE 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RX_DONE 中断被屏蔽		
	0x1 → 已禁用: RX_DONE 中断未被屏蔽		
6	M_TX_ABRT: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_ABRT 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_ABORT 中断被屏蔽		
	0x1 → 已禁用: TX_ABORT 中断未被屏蔽		
5	M_RD_REQ: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RD_REQ 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RD_REQ 中断被屏蔽		
	0x1 → 已禁用: RD_REQ 中断未被屏蔽		
4	M_TX_EMPTY: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_EMPTY 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_EMPTY 中断被屏蔽		
	0x1 → 已禁用: TX_EMPTY 中断未被屏蔽		

位	描述	类型	复位值
3	M_TX_OVER: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_OVER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_OVER 中断被屏蔽		
	0x1 → 已禁用: TX_OVER 中断未被屏蔽		
2	M_RX_FULL: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_FULL 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RX_FULL 中断被屏蔽		
	0x1 → 已禁用: RX_FULL 中断未被屏蔽		
1	M_RX_OVER: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_OVER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 启用: RX_OVER 中断被屏蔽		
	0x1 → 禁用: RX_OVER 中断未被屏蔽		
0	M_RX_UNDER: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_UNDER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 启用: RX_UNDER 中断被屏蔽		
	0x1 → 禁用: RX_UNDER 中断未被屏蔽		

I2C: IC_RAW_INTR_STAT 寄存器

偏移: 0x34

描述

I2C原始中断状态寄存器

不同于 IC_INTR_STAT 寄存器，此处的位未被屏蔽，因此始终显示 DW_apb_i2c 的真实状态。

表 463。
IC_RAW_INTR_STAT
寄存器

位	描述	类型	复位值
31:13	保留。	-	-

位	描述	类型	复位值
12	<p>RESTART_DET: 指示在 DW_apb_i2c 作为从机模式且从机被寻址时, I2C 接口是否发生了 RESTART 条件。仅当 IC_SLV_RESTART_DET_EN=1 时启用。</p> <p>注意: 但根据I2C协议, 在高速模式或传输START BYTE期间, RESTART 应位于地址字段之前。在此情况下, 当发出RESTART时, 被寻址的从机并非当前从机, 因此DW_apb_i2c不会产生RESTART_DET中断。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: RESTART_DET 中断处于非活动状态		
	0x1 → 活动: RESTART_DET 中断处于活动状态		
11	<p>GEN_CALL: 仅在接收到通用呼叫地址并确认时设置。该标志保持设置, 直到通过禁用DW_apb_i2c或CPU读取IC_CLR_GEN_CALL寄存器的第0位将其清除。DW_apb_i2c将接收的数据存储于接收缓冲区 (Rx buffer) 中。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: GEN_CALL 中断处于非活动状态		
	0x1 → 活动: GEN_CALL 中断处于活动状态		
10	<p>START_DET: 指示无论DW_apb_i2c处于从机还是主机模式, I2C接口上是否出现START或RESTART条件。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: START_DET 中断处于非活动状态		
	0x1 → 活动: START_DET 中断处于活动状态		
9	<p>STOP_DET: 指示无论 DW_apb_i2c 处于从模式还是主模式, I2C 接口上是否发生了停止条件。</p> <p>在从模式下: - 当 IC_CON[7]=1'b1 (STOP_DET_IFADDRESSED) 时, 仅在从设备被寻址时触发 STOP_DET 中断。注意: 在通用调用地址期间, 若 STOP_DET_IF_ADDRESSED=1'b1, 即使从设备通过响应通用调用地址并产生 ACK, 从设备也不会触发 STOP_DET 中断。仅当传输地址与从设备地址 (SAR) 匹配时, 才会触发 STOP_DET 中断。- 当 IC_CON[7]=1'b0 (STOP_DET_IFAD DRESSED) 时, 无论是否被寻址, 均会触发 STOP_DET 中断。在主模式下: - 当 IC_CON[10]=1'b1 (STOP_DET_IF_MASTER_ACTIVE) 时, 仅当主设备处于活动状态时才触发 STOP_DET 中断。- 若 IC_CON[10]=1'b0 (STOP_ DET_IFADDRESSED) , 则无论主控是否处于活动状态, 均会触发 ST OP_DET 中断。复位值: 0x0</p>	只读	0x0
	枚举值:		

位	描述	类型	复位值
	0x0 → 非活动：STOP_DET 中断处于非活动状态		
	0x1 → 活动：STOP_DET 中断处于活动状态		
8	<p>ACTIVITY：该位记录 DW_apb_i2c 活动信号，并保持置位直至被清除。清除该位的方法有四种：</p> <ul style="list-style-type: none"> - 禁用 DW_apb_i2c - 读取 IC_CLR_ACTIVITY 寄存器 - 读取 IC_CLR_INTR 寄存器 - 系统复位 - 一旦该位被置位，除非通过上述任一方法进行清除，否则将持续保持置位。即便 DW_apb_i2c 模块空闲，该位亦表明总线曾有活动且保持置位状态。 <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：RAW_INTR_ACTIVITY 中断处于非活动状态		
	0x1 → 活动：RAW_INTR_ACTIVITY 中断处于活动状态		
7	<p>RX_DONE：当 DW_apb_i2c 作为从设备发送方时，如果主设备未确认所发送的字节，该位将被置为1。此状态发生在传输的最后一个字节，表示传输已完成。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：RX_DONE 中断处于非活动状态。		
	0x1 → 活动：RX_DONE 中断处于活动状态。		
6	<p>TX_ABRT：该位指示 DW_apb_i2c 作为 I2C 发送方时，无法完成对发送 FIFO 内容的预期操作。此情况可能发生在 I2C 主设备或从设备，称为“发送中止”。当该位被置为1时，IC_TX_ABRT_SOURCE 寄存器会指示发送中止的具体原因。</p> <p>注意：无论导致发送中止的事件属于 IC_TX_ABRT_SOURCE 寄存器所跟踪的哪种情况，DW_apb_i2c 都会清空、重置或清除 TX_FIFO 和 RX_FIFO。FIFO 将保持在此清空状态，直到读取寄存器 IC_CLR_TX_ABRT。一旦执行该读取操作，Tx FIFO 即准备好接受来自 APB 接口的更多数据字节。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非激活：TX_ABRT 中断处于非激活状态		
	0x1 → 激活：TX_ABRT 中断处于激活状态		
5	<p>RD_REQ：当 DW_apb_i2c 作为从设备且另一 I2C 主设备试图从 DW_apb_i2c 读取数据时，该位被置为1。DW_apb_i2c 会将 I2C 总线保持在等待状态 (SC L=0)，直到该中断被处理，这意味着从设备已被远程主设备寻址，且主设备请求进行数据传输。处理器必须响应该中断，然后将请求的数据写入 IC_DARTA_CMD 寄存器。该位在处理器读取 IC_CLR_RD_REQ 寄存器后立即被清零。</p> <p>复位值：0x0</p>	只读	0x0

位	描述	类型	复位值
	枚举值：		
	0x0 → 非活动： RD_REQ 中断处于非活动状态		
	0x1 → 活动： RD_REQ 中断处于活动状态		
4	<p>TX_EMPTY: TX_EMPTY 中断状态的行为因 IC_CON 寄存器中 TX_EMPTY_C_TRL 的设置而异。 - 当 TX_EMPTY_CTRL = 0 时：当发射缓冲区中数据量小于或等于 IC_TX_TL 寄存器中设置的阈值时，该位被置为 1 。 - 当 TX_EMPTY_CTRL = 1 时：当发射缓冲区中数据量小于或等于 IC_TX_TL 寄存器中设置的阈值，且最近出栈命令的内部移位寄存器中的地址/数据传输完成时，该位被置为 1。当缓冲区数据量高于阈值时，硬件会自动清除此位。当 IC_ENABLE[0] 设置为 0 时，TX FIFO 会被清空并保持复位状态。此时，TX FIFO 似乎无数据，在主机或从机状态机存在活动时，该位被置为 1。当不再有任何活动且 ic_en=0 时，此位被置为 0。</p> <p>复位值：0x0。</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： TX_EMPTY 中断处于非活动状态		
	0x1 → 活动： TX_EMPTY 中断处于活动状态		
3	<p>TX_OVER: 当传输缓冲区填满至 IC_TX_BUFFER_DEPTH 且处理器尝试通过写入 IC_DATA_CMD 寄存器发出另一个 I2C 命令时，在传输过程中设置该位。当模块被禁用时，此位保持其状态，直到主机或从机状态机进入空闲状态；当 ic_en 变为 0 时，此中断被清除。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： TX_OVER 中断处于非活动状态		
	0x1 → 活动： TX_OVER 中断处于活动状态		
2	<p>RX_FULL: 当接收缓冲区达到或超过 IC_RX_TL 寄存器中 RX_TL 阈值时设置。当缓冲区电平降至阈值以下时，硬件会自动清除此位。如果模块被禁用 (IC_ENABLE[0]=0) ，则 RX FIFO 将被清空并保持复位状态；因此 RX FIFO 未满。因此，一旦 IC_ENABLE 位 0 被设置为 0，无论后续活动如何，该位即被清除。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： RX_FULL 中断处于非激活状态		
	0x1 → 激活： RX_FULL 中断处于激活状态		

位	描述	类型	复位值
1	<p>RX_OVER: 当接收缓冲区完全填满, 且达到 IC_RX_BUFFER_DEPTH 后, 再从外部 I2C 设备接收额外字节时设置。DW_apb_i2c 会对此进行应答, 但 FIFO 满后接收的任何数据字节将丢失。如果模块被禁用 (IC_ENABLE[0]=0), 该位保持其状态, 直至主机或从机状态机进入空闲状态, 并且当 ic_en 变为 0 时, 该中断被清除。</p> <p>注意: 若 IC_CON 寄存器的第 9 位 (RX_FIFO_FULL_HLD_CTRL) 设为高电平, 则 RX_OVER 中断不会发生, 因为 Rx FIFO 永远不会溢出。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非激活: RX_OVER 中断处于非激活状态		
	0x1 → 激活: RX_OVER 中断处于激活状态		
0	<p>RX_UNDER: 当处理器尝试通过读取 IC_DATA_CMD 寄存器读取空的接收缓冲区时设置该标志。如果模块被禁用 (IC_ENABLE[0]=0), 该位保持其状态, 直至主机或从机状态机进入空闲状态, 并且当 ic_en 变为 0 时, 该中断被清除。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: RX_UNDER 中断处于非活动状态		
	0x1 → 活动: RX_UNDER 中断处于活动状态		

I2C: IC_RX_TL 寄存器

偏移: 0x38

描述

I2C接收FIFO阈值寄存器

表 464. IC_RX_TL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	<p>RX_TL: 接收 FIFO 阈值等级。</p> <p>控制触发 RX_FULL 中断 (IC_RAW_INTR_STAT 寄存器第 2 位) 所需的条目数 (及以上)。有效取值范围为 0-255, 且硬件不允许将该值设置为超过缓冲区深度的数值。若尝试设置超过缓冲区深度的值, 实际生效值将为缓冲区的最大深度。取值为 0 时, 阈值为 1 个条目; 取值为 255 时, 阈值为 256 个条目。</p>	读写	0x00

I2C: IC_TX_TL 寄存器

偏移: 0x3c

描述

I2C发送FIFO阈值寄存器

表465. IC_TX_TL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TX_TL : 发送FIFO阈值级别。 控制触发TX_EMPTY中断 (IC_RAW_INTR_STAT寄存器第4位) 的条目数阈值 (等于或低于该阈值)。有效范围为0至255, 且不可设置为超过缓冲区深度的值。若尝试设置超过缓冲区深度的值, 实际生效值将为缓冲区的最大深度。设置为0表示阈值为0条目, 设置为255表示阈值为255条目。	读写	0x00

I2C: IC_CLR_INTR寄存器

偏移: 0x40

描述

清除组合中断及单个中断寄存器

表466.
IC_CLR_INTR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_INTR : 读取此寄存器以清除组合中断、所有单个中断, 以及IC_TX_ABRT_SOURCE寄存器。此位不会清除硬件可清除的中断, 仅清除软件可清除的中断。清除IC_TX_ABRT_SOURCE的例外情况详见IC_TX_ABRT_SOURCE寄存器的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_UNDER寄存器

偏移: 0x44

描述

清除 RX_UNDER 中断寄存器

表467.
IC_CLR_RX_UNDER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RX_UNDER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_UNDER 中断 (第0位)。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_OVER寄存器

偏移: 0x48

描述

清除 RX_OVER 中断寄存器

表468.
IC_CLR_RX_OVER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-

位	描述	类型	复位值
0	CLR_RX_OVER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_OVER 中断（第1位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_TX_OVER 寄存器

偏移量: 0x4c

说明

清除 TX_OVER 中断寄存器

表469。
IC_CLR_TX_OVER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_TX_OVER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 TX_OVER 中断（第3位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RD_REQ 寄存器

偏移: 0x50

说明

清除 RD_REQ 中断寄存器

表470。
IC_CLR_RD_REQ
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RD_REQ : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RD_REQ 中断（第5位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_TX_ABRT 寄存器

偏移: 0x54

说明

清除 TX_ABRT 中断寄存器

表471。
IC_CLR_TX_ABRT
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_TX_ABRT : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 TX_ABRT 中断（第6位）及 IC_TX_ABRT_SOURCE 寄存器的状态。这也会解除 TX FIFO 的刷新/复位状态，允许向 TX FIFO 进行更多写入。有关不清除 IC_TX_ABRT_SOURCE 的例外情况，请参阅 IC_TX_ABRT_SOURCE 寄存器的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_DONE 寄存器

偏移: 0x58

说明

清除 RX_DONE 中断寄存器

表472。
IC_CLR_RX_DONE
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RX_DONE : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_DONE 中断（第7位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_ACTIVITY 寄存器

偏移: 0x5c

描述

清除 ACTIVITY 中断寄存器

表473。
IC_CLR_ACTIVITY
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_ACTIVITY : 读取此寄存器以清除当 I2C 总线不再活动时的 ACTIVITY 中断。若 I2C 模块仍在总线上处于活动状态，ACTIVITY 中断位将继续置位。该中断位在模块被禁用且总线上无进一步活动时，会由硬件自动清除。读取此寄存器可获取 IC_RAW_INTR_STAT 寄存器中 ACTIVITY 中断（第8位）的状态。 复位值: 0x0	只读	0x0

I2C: IC_CLR_STOP_DET 寄存器

偏移: 0x60

描述

清除 STOP_DET 中断寄存器

表474。
IC_CLR_STOP_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-

位	描述	类型	复位值
0	CLR_STOP_DET : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 STOP_DET 中断的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_START_DET 寄存器

偏移: 0x64

描述

清除 START_DET 中断寄存器

表475。
IC_CLR_START_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_START_DET : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 START_DET 中断的第10位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_GEN_CALL 寄存器

偏移: 0x68

描述

清除 GEN_CALL 中断寄存器

表476。
IC_CLR_GEN_CALL
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_GEN_CALL : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 GEN_CALL 中断的第11位。 复位值: 0x0	只读	0x0

I2C: IC_ENABLE 寄存器

偏移: 0x6c

描述

I2C 使能寄存器

表477。IC_ENABLE
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	TX_CMD_BLOCK : 在主机模式下: - 1'b1: 即使 Tx FIFO 中有数据待传输, 也阻止在 I2C 总线上发送数据。- 1'b0: 数据传输在 I2C 总线上自动启动, 当 Tx FIFO 中有第一个数据可用时即刻开始。注意: 仅当 Tx FIFO 为空 (IC_STATUS[2] == 1) 且主机处于空闲状态 (IC_STATUS[5] == 0) 时, 方可设置 TX_CMD_BLOCK 位以阻止主机命令执行。之后放入 Tx FIFO 的任何命令均不会被执行, 直至清除 TX_CMD_BLOCK 位。复位值为: IC_TX_CMD_BLOCK_DEFAULT	读写	0x0
	枚举值:		
	0x0 → NOT_BLOCKED: Tx 命令执行未被阻止		

位	描述	类型	复位值
	0x1 → BLOCKED: Tx命令执行已被阻止		
1	<p>ABORT: 设置该位后，控制器启动传输中止操作。- 0：未启动中止或中止已完成； - 1：中止操作正在进行中。软件可通过设置该位在主模式下中止I2C传输。仅当ENABLE位已设置时，软件方可设置该位；否则控制器将忽略对ABORT位的任何写入。一旦设置，软件无法清除该位。响应中止时，控制器将在完成当前传输后发出STOP信号并清空Tx FIFO，随后触发TX_ABORT中断。ABORT 位在中止操作完成后会自动清除。</p> <p>有关如何中止 I2C 传输的详细说明，请参阅“中止 I2C 传输”。</p> <p>复位值：0x0</p>	读写	0x0
	枚举值：		
	0x0 → 禁用：未进行 ABORT 操作		
	0x1 → 启用：正在进行 ABORT 操作		
0	<p>ENABLE: 控制是否启用 DW_apb_i2c。 - 0：禁用 DW_apb_i2c (TX 和 RX FIFO 保持在擦除状态) - 1：启用 DW_apb_i2c。软件可在 DW_apb_i2c 处于活动状态时禁用它。然而，务必谨慎确保 DW_apb_i2c 被正确禁用。推荐的操作步骤详见“禁用 DW_apb_i2c”。</p> <p>当 DW_apb_i2c 被禁用时，发生以下情况： - TX FIFO 和 RX FIFO 会被清空。 - IC_INTR_STAT 寄存器中的状态位仍然有效，直到 DW_apb_i2c 进入空闲 (I DLE) 状态。如果模块正在传输，则在当前传输完成后停止传输并清空传输缓冲区内容。如果模块正在接收，DW_apb_i2c 会在当前字节结束时停止传输且不确认该次传输。</p> <p>在 pclk 与 ic_clk 异步的系统中，当 IC_CLK_TYPE 参数设置为异步 (1) 时，启用或禁用 DW_apb_i2c 会延迟两个 ic_clk 时钟周期。有关如何禁用 DW_apb_i2c 的详细说明，请参见“禁用 DW_apb_i2c”一节。</p> <p>复位值：0x0</p>	读写	0x0
	枚举值：		
	0x0 → DISABLED：I2C 已禁用		
	0x1 → ENABLED：I2C 已启用		

I2C: IC_STATUS 寄存器

偏移：0x70

描述

I2C 状态寄存器

该寄存器为只读，用于指示当前传输状态及 FIFO 状态。状态寄存器可在任意时刻读取。该寄存器中的所有位均未请求中断。

当通过向IC_ENABLE寄存器的第0位写入0以禁用I2C时： - 第1位和第2位被置为1 - 第3位和第10位被置为0 当主设备或从设备状态机进入空闲且ic_en=0时： - 第5位和第6位被置为0

表478. IC_STATUS寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6	SLV_ACTIVITY : 从机有限状态机 (FSM) 的活动状态。当从机有限状态机不处于IDLE状态时，该位被置位。- 0: 从机FSM处于IDLE状态，DW_apb_i2c的从机部分处于非活动状态 - 1: 从机FSM不处于IDLE状态，DW_apb_i2c的从机部分处于活动状态。复位值: 0x0 枚举值： 0x0 → IDLE：从机空闲 0x1 → ACTIVE：从机非空闲	只读	0x0
5	MST_ACTIVITY : 主控有限状态机 (FSM) 活动状态。当主控有限状态机不处于IDLE状态时，该位被置位。- 0: 主控FSM处于IDLE状态，DW_apb_i2c的主控部分处于非活动状态 - 1: 主控FSM不处于IDLE状态，DW_apb_i2c的主控部分处于活动状态。注意： IC_STATUS[0]——即ACTIVITY位——是SLV_ACTIVITY和MST_ACTIVITY位的逻辑或。 复位值: 0x0 枚举值： 0x0 → IDLE：主控空闲 0x1 → ACTIVE：主控不空闲	只读	0x0
4	RFF : 接收FIFO完全满。当接收FIFO完全满时，该位被置位；当接收FIFO有一个或多个空位时，该位被清除。- 0: 接收FIFO未满 - 1: 接收FIFO已满。复位值: 0x0 枚举值： 0x0 → NOT_FULL：接收FIFO未满 0x1 → FULL：接收FIFO已满	只读	0x0
3	RFNE : 接收FIFO非空。当接收FIFO中有一条或多条数据时，该位被置位；当接收FIFO为空时，该位被清除。- 0: 接收FIFO为空 - 1: 接收FIFO非空 复位值: 0x0 枚举值： 0x0 → EMPTY：接收FIFO为空 0x1 → NOT_EMPTY：接收FIFO非空	只读	0x0
2	TFE : 发送FIFO完全为空。当发送FIFO完全为空时，该位被置位；只要包含一条或多条有效数据，该位即被清除。该位域不请求中断。- 0: 发送FIFO非空 - 1: 发送FIFO为空 复位值: 0x1 枚举值： 0x0 → NON_EMPTY：发送FIFO非空 0x1 → EMPTY：发送FIFO为空	只读	0x1

位	描述	类型	复位值
1	TFNF : 发送FIFO未满。当发送FIFO有一个或多个空位时，该位被置位；当FIFO已满时，该位被清除。- 0: 发送FIFO已满 - 1: 发送FIFO未满 复位值: 0x1	只读	0x1
	枚举值:		
	0x0 → FULL: 发送FIFO已满		
	0x1 → NOT_FULL: 发送FIFO未满		
0	ACTIVITY : I2C活动状态。复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → INACTIVE: I2C处于空闲状态		
	0x1 → ACTIVE: I2C处于活动状态		

I2C: IC_TXFLR寄存器

偏移: 0x74

描述

I2C发送FIFO级别寄存器 该寄存器包含发送FIFO缓冲区中有效数据项的数量。

当满足以下条件时清除: - 禁用 I2C - 发生传输中止, 即 IC_RAW_INTR_STAT 寄存器中的 TX_ABRT 位被置位 - 从机批量传输模式中止。每当数据被放入传输 FIFO 时, 寄存器递增; 当数据从传输 FIFO 取出时, 寄存器递减。

表 479. IC_TXFLR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	TXFLR : 发送FIFO级别。表示发送FIFO中有效数据条目的数量。 复位值: 0x0	只读	0x00

I2C: IC_RXFLR寄存器

偏移: 0x78

描述

I2C接收FIFO级别寄存器。该寄存器表示接收FIFO缓冲区中有效数据条目的数量。该寄存器在以下情况下清零: - I2C 被禁用 - 由IC_TX_ABRT_SOURCE中跟踪的任一事件导致的发送中止。每当数据写入接收FIFO, 寄存器值递增; 每当数据从接收FIFO取出, 寄存器值递减。

表 480. IC_RXFLR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	RXFLR : 接收FIFO级别。表示接收FIFO中有效数据条目的数量。 复位值: 0x0	只读	0x00

I2C: IC_SDA_HOLD寄存器

偏移: 0x7c

描述

I2C SDA 保持时间寄存器

该寄存器的[15:0]位用于控制在主从模式下传输过程中SDA信号的保持时间
(在SCL由高电平变为低电平后)。

该寄存器的[23:16]位用于在主从模式的接收端，当SCL为高电平时，延长SDA信号的过渡时间（如有）。

仅当 IC_ENABLE[0]=0 时，写入该寄存器操作方可成功。

该寄存器中的数值以 ic_clk 周期为单位。IC_SDA_RX_HOLD 中编程的值必须大于各模式下的最小保持时间（主模式为一个周期，从模式为七个周期），该值方能生效。

传输期间编程的 SDA 保持时间 (IC_SDA_TX_HOLD) 任意时刻均不得超过 scl 低电平的持续时间。因此，编程值不得大于 N_SCL_LOW-2，其中 N_SCL_LOW 为以 ic_clk 周期计量的 scl 低电平持续时间。

表 481。
IC_SDA_HOLD
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	IC_SDA_RX_HOLD : 设置当 DW_apb_i2c 作为接收器时所需的 SDA 保持时间，单位为 ic_clk 周期。 复位值：IC_DEFAULT_SDA_HOLD[23:16]。	读写	0x00
15:0	IC_SDA_TX_HOLD : 设置当 DW_apb_i2c 作为发送器时所需的 SDA 保持时间，单位为 ic_clk 周期。 复位值：IC_DEFAULT_SDA_HOLD[15:0]。	读写	0x0001

I2C: IC_TX_ABRT_SOURCE 寄存器

偏移: 0x80

描述

I2C 发送中止源寄存器

该寄存器包含32位，用于指示TX_ABRT位的来源。除第9位外，每当读取IC_CLR_TX_ABRT寄存器或IC_CLR_INTR寄存器时，该寄存器即被清除。要清除第9位，必须先修正ABRT_SBYTE_NORSTRT的来源；必须启用RESTART (IC_CON[5]=1)，且SPECIAL位 (IC_TAR[11]) 必须被清除，或者GC_OR_START位 (IC_TAR[10]) 必须被清除。

一旦修正了ABRT_SBYTE_NORSTRT的来源，即可用本寄存器其他位相同的方式清除此位。若在尝试清除此位之前未修正 ABRT_SBYTE_NORSTRT 的来源，第9位将仅清除一个周期，随后再次被置位。

表 482。
IC_TX_ABRT_SOURCE
寄存器

位	描述	类型	复位值
31:23	TX_FLUSH_CNT : 此字段指示因 TX_ABRT 中断而被清空的 Tx FIFO 数据命令数量。当 I2C 被禁用时，该字段将被清零。 复位值：0x0 DW_apb_i2c 角色：主机发送器或从机发送器	只读	0x000
22:17	保留。	-	-

位	描述	类型	复位值
16	<p>ABRT_USER_ABRT: 该位仅适用于主模式。主机检测到传输中止 (IC_ENABLE[1])</p> <p>复位值: 0x0</p> <p>DW_apb_i2c 角色: 主机发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_USER_ABRT_VOID: 主机检测到的传输中止场景不存在		
	0x1 → ABRT_USER_ABRT_GENERATED: 主机检测到的传输中止		
15	<p>ABRT_SLVRD_INTX: 1: 当处理器侧响应从模式请求, 向远程主机传输数据, 且用户在 IC_DATA_CMD 寄存器的 CMD (第8位) 写入1时。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SLVRD_INTX_VOID: 从机尝试在读取模式下向远程主机发送数据——此情景不存在		
	0x1 → ABRT_SLVRD_INTX_GENERATED: 从机尝试在读取模式下向远程主机发送数据		
14	<p>ABRT_SLV_ARBLOST: 该字段指示从机在向远程主机传输数据时丢失总线控制权。同时设置了IC_TX_ABRT_SOURCE[12]。注意: 尽管从机从未“拥有”总线, 但总线上仍可能发生故障。本项为故障保护检测。例如, 在SCL由低到高的传输过程中, 如果数据总线上的数据与应传输的数据不符, 则DW_apb_i2c即不再拥有该总线。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SLV_ARBLOST_VOID: 从机在仲裁中败给远程主机——方案不存在		
	0x1 → ABRT_SLV_ARBLOST_GENERATED: 从机在仲裁中败给远程主设备		
13	<p>ABRT_SLVFLUSH_TXFIFO: 该字段表示从机已接收到读取命令且TX FIFO中存在数据, 因此从机发出TX_ABRT中断以清除TX FIFO中的旧数据。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		

位	描述	类型	复位值
	0x0 → ABRT_SLVFLUSH_TXFIFO_VOID：从机在收到读取命令时清除TX FIFO 中现有数据——方案不存在		
	0x1 → ABRT_SLVFLUSH_TXFIFO_GENERATED：从机在收到读取命令时清除 TX FIFO 中现有数据		
12	<p>ARB_LOST：该字段表示主设备失去仲裁，或在IC_TX_ABRT_SOURCE[1:4]置位时表示从机发送器失去仲裁。</p> <p>复位值：0x0</p> <p>DW_apb_i2c 角色：主机发送器或从机发送器</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_LOST_VOID：主设备或从机发送器失去仲裁——方案不存在		
	0x1 → ABRT_LOST_GENERATED：主机或从机发送端仲裁丢失		
11	<p>ABRT_MASTER_DIS：该字段指示用户在主机模式禁用时尝试启动主机操作。</p> <p>复位值：0x0</p> <p>DW_apb_i2c的角色：主机发送端或主机接收端</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_MASTER_DIS_VOID：用户在主机禁用时启动主机操作的情形不存在		
	0x1 → ABRT_MASTER_DIS_GENERATED：用户在主机禁用时启动主机操作		
10	<p>ABRT_10B_RD_NORSTRT：该字段表示重启功能被禁用 (IC_RESTART_EN位 (IC_CON[5]) =0) ，且主机在10位地址模式下发送读取命令。</p> <p>复位值：0x0</p> <p>DW_apb_i2c的角色：主机接收端</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_10B_RD_VOID：当重启功能禁用时，主机未尝试在10位地址模式下读取		
	0x1 → ABRT_10B_RD_GENERATED：主控设备在RESTART禁用时尝试以10位寻址模式读取		

位	描述	类型	复位值
9	<p>ABRT_SBYTE_NORSTRT: 要清除第9位，必须先修正ABRT_SBYTE_NORSTRT的根本原因；必须启用restart (IC_CON[5]=1)，清除SPECIAL位 (IC_TAR[11])，或清除GC_OR_START位 (IC_TAR[10])。一旦修正ABRT_SBYTE_NORSTRT的根本原因，该位即可按此寄存器中其他位的相同方式清除。若未修正ABRT_SBYTE_NORSTRT的根本原因而试图清除此位，则第9位会被清除一个周期后重新置位。当该字段设置为1时，restart被禁用 (IC_RESTART_EN位(IC_CON[5])=0)，且用户正在尝试发送START字节。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SBYTE_NORSTRT_VOID : 用户尝试在禁用重启 (RESTART) 时发送 START 字节——该场景不存在		
	0x1 → ABRT_SBYTE_NORSTRT_GENERATED : 用户尝试在禁用重启 (RESTART) 时发送 START 字节		
8	<p>ABRT_HS_NORSTRT: 该字段指示重启功能被禁用 (IC_RESTART_EN 位 (IC_CON[5]) =0) 且用户尝试使用主控以高速模式传输数据。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主机发送端或主机接收端</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_HS_NORSTRT_VOID : 用户尝试在禁用重启 (RESTART) 时切换主控至高速模式——该场景不存在		
	0x1 → ABRT_HS_NORSTRT_GENERATED : 用户尝试在禁用重启 (RESTART) 时切换主控至高速模式		
7	<p>ABRT_SBYTE_ACKDET: 该字段指示主控已发送 START 字节且该 START 字节被应答（错误行为）。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SBYTE_ACKDET_VOID : 检测到START字节的ACK——该情况不存在		
	0x1 → ABRT_SBYTE_ACKDET_GENERATED : 检测到START字节的ACK		
6	<p>ABRT_HS_ACKDET: 该字段表示主机处于高速模式且高速主机代码已被确认（为不当行为）。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0

位	描述	类型	复位值
	枚举值：		
	0x0 → ABRT_HS_ACK_VOID：高速模式下高速主机代码被确认——该情况不存在		
	0x1 → ABRT_HS_ACK_GENERATED：高速模式下高速主机代码被确认		
5	ABRT_GCALL_READ : 该字段表示DW_apb_i2c处于主机模式时发送了广播呼叫，但用户将广播呼叫后的字节编程为从总线读取 (IC_DATA_CMD[9] 设置为1)。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_GCALL_READ_VOID：广播呼叫后接从总线读取——该情况不存在		
	0x1 → ABRT_GCALL_READ_GENERATED：GCALL 后紧跟对总线的读取		
4	ABRT_GCALL_NOACK : 该字段指示 DW_apb_i2c 主模式已发送 General Call，但总线上的任何从设备均未确认该 General Call。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_GCALL_NOACK_VOID：未有任何从设备确认 GCALL 的情况不存在		
	0x1 → ABRT_GCALL_NOACK_GENERATED：未有任何从设备确认 GCALL		
3	ABRT_TXDATA_NOACK : 该字段为仅限主模式位。当主设备接收到地址确认后，发送后续数据字节时未收到远程从设备确认。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_TXDATA_NOACK_VOID：发送的数据未被指定从设备确认的情况不存在		
	0x1 → ABRT_TXDATA_NOACK_GENERATED：发送的数据未被指定从设备确认		
2	ABRT_10ADDR2_NOACK : 该字段指示主机处于10位地址模式，且10位地址的第二字节未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色: 主机发送端或主机接收端	只读	0x0

位	描述	类型	复位值
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：10位地址的第2字节未被任何从机确认		
1	ABRT_10ADDR1_NOACK : 该字段指示主机处于10位地址模式，且第一字节10位地址未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色：主机发送端或主机接收端	只读	0x0
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：10位地址的第1字节未被任何从机确认		
0	ABRT_7B_ADDR_NOACK : 该字段指示主机处于7位地址模式，且发送的地址未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色：主机发送端或主机接收端	只读	0x0
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：因7位地址未响应（NOACK）而生成此中止信号		

I2C: IC_SLV_DATA_NACK_ONLY 寄存器

偏移: 0x84

描述

生成从设备数据 NACK 寄存器

该寄存器用于在DW_apb_i2c作为从机接收器时，为传输的数据段生成NACK。

仅当IC_SLV_DATA_NACK_ONLY参数设置为1时，该寄存器才存在。当该参数被禁用时，该寄存器不存在，写入该寄存器地址不会产生任何效果。

仅当满足以下两个条件时，才能对该寄存器进行写入： - DW_apb_i2c 被禁用 (IC_ENABLE[0] = 0) - 从机部分处于非活动状态 (IC_STATUS[6] = 0)
注：IC_STATUS[6] 是内部 slv_activity 信号的寄存器读取反馈位；用户应在写入 ic_slv_data_nack_only 位之前，先轮询该位。

表 483。
IC_SLV_DATA_NACK_ONLY 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	NACK : 生成 NACK。该 NACK 仅在 DW_apb_i2c 作为从机接收器时生成。如果该寄存器设置为1，则仅在接收到数据字节后生成NACK；因此，数据传输被中止，接收的数据不会被推入接收缓冲区。 当寄存器设置为0时，根据正常条件生成NACK或ACK。- 1: 接收数据字节后生成NACK - 0: 正常生成NACK或ACK 重置值: 0x0	读写	0x0
	枚举值：		

位	描述	类型	复位值
	0x0 → 禁用：从机接收器正常生成NACK		
	0x1 → 启用：从机接收器仅在接收数据时生成NACK		

I2C: IC_DMA_CR 寄存器

偏移量: 0x88

描述

DMA 控制寄存器

该寄存器用于使能 DMA 控制器接口操作。发送和接收各自拥有独立控制位。

可在 IC_ENABLE 状态下任意编程。

表 484。
IC_DMA_CR 寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TDMAE : 传输DMA使能。此位用于启用或禁用传输FIFO DMA通道。复位值： : 0x0	读写	0x0
	枚举值：		
	0x0 → 禁用：传输FIFO DMA通道被禁用		
	0x1 → 启用：传输FIFO DMA通道被启用		
0	RDMAE : 接收DMA使能。此位用于启用或禁用接收FIFO DMA通道。复位值： 0x0	读写	0x0
	枚举值：		
	0x0 → 禁用：接收FIFO DMA通道被禁用		
	0x1 → 启用：接收FIFO DMA通道被启用		

I2C: IC_DMA_TDRL 寄存器

偏移量: 0x8c

说明

DMA 传输数据级别寄存器

表 485。
IC_DMA_TDRL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	DMATDL : 传输数据级别。该字段控制传输逻辑触发DMA请求的阈值。其值等同于水位线；即当传输FIFO中有效数据条目数小于或等于该字段值且 TDMAE=1 时，dma_tx_req信号被激活。 复位值：0x0	读写	0x0

I2C: IC_DMA_RDRL 寄存器

偏移：0x90

描述

I2C 接收数据水位寄存器

表 486。
IC_DMA_RDLR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	DMARDL : 接收数据水位。此字段控制接收逻辑发送DMA请求的水位。水位线 = DMARDL + 1; 即当接收FIFO中有效数据条目数大于或等于该字段值加1且RDMAE = 1时, dma_rx_req信号被触发。例如, 当DMARDL为0时, 接收FIFO中存在1条或以上数据时, 将触发dma_rx_req。 复位值: 0x0	读写	0x0

I2C: IC_SDA_SETUP 寄存器

偏移: 0x94

描述

I2C SDA 设置寄存器

本寄存器控制DW_apb_i2c在从机作为发射端响应读请求时, SCL上升沿相对于SDA变化所引入的延迟时间（以ic_clk时钟周期计）。相关的I2C要求为tSU:DAT（注4），详见I2C总线规格。本寄存器必须编程为大于或等于2的值。

仅当IC_ENABLE[0] = 0时, 写入此寄存器操作才会成功。

注：设置时间的长度通过[(IC_SDA_SETUP - 1) * (ic_clk_period)]计算，因此若用户需要10个ic_clk周期的设置时间，应编程为11。IC_SDA_SETUP寄存器仅在DW_apb_i2c作为从机发送时使用。

表 487。
IC_SDA_SETUP
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SDA_SETUP : SDA设置。建议当所需延迟为1000纳秒且ic_clk频率为10 MHz时, IC_SDA_SETUP应编程为11。IC_SDA_SETUP寄存器的最小编程值为2。	读写	0x64

I2C: IC_ACK_GENERAL_CALL 寄存器

偏移: 0x98

说明

I2C 应答通用调用寄存器

该寄存器用于控制 DW_apb_i2c 接收到 I2C 通用呼叫地址时, 响应 ACK 还是 NACK。

该寄存器仅在 DW_apb_i2c 处于从模式时适用。

表 488。
IC_ACK_GENERAL_CA
LL 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	ACK_GEN_CALL : ACK 通用呼叫。设置为 1 时, DW_apb_i2c 在接收到通用呼叫时通过置位 ic_data_oe 响应 ACK; 否则, 通过清除 ic_data_oe 响应 NACK。 枚举值: 0x0 → 禁用: 对通用呼叫生成 NACK 0x1 → 启用: 对通用呼叫生成 ACK	读写	0x1

I2C: IC_ENABLE_STATUS 寄存器

偏移: 0x9c

描述

I2C 使能状态寄存器

当 IC_ENABLE[0] 寄存器由 1 变更为 0 时，该寄存器用于报告 DW_apb_i2c 硬件状态；即，当 DW_apb_i2c 被禁用时。

如果 IC_ENABLE[0] 被设置为 1，则位 2:1 被强制设为 0，且位 0 被强制设为 1。

如果 IC_ENABLE[0] 被设置为 0，则仅当位 0 读取为“0”时，位 2:1 才有效。

注意：当 IC_ENABLE[0] 被设置为 0 时，位 0 读取为 0 会出现延迟，因为禁用 DW_apb_i2c 依赖于 I2C 总线活动。

表 489。
IC_ENABLE_STATUS
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	<p>SLV_RX_DATA_LOST: 从机接收数据丢失。该位指示当 IC_ENABLE 位 0 从 1 变为 0 时，从机接收操作因至少接收到一个 I2C 传输的数据字节而被中止。当读取为 1 时，DW_apb_i2c 被视为已主动参与被中止的 I2C 传输（匹配地址），且已进入 I2C 传输的数据阶段，即使数据字节已被响应为 NACK。</p> <p>注意：如果远程 I2C 主机在 DW_apb_i2c 有机会对传输发送 NACK 之前以 STOP 条件终止传输，且 IC_ENABLE[0] 已设置为 0，则该位亦被置为 1。</p> <p>当读取值为 0 时，DW_apb_i2c 被视为已禁用，且未主动介入从机接收传输的数据阶段。</p> <p>注意：仅当 IC_EN（第 0 位）读取为 0 时，CPU 才可安全读取此位。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：从机接收数据未丢失		
	0x1 → 活动：从机接收数据丢失		

位	描述	类型	复位值
1	<p>SLV_DISABLED_WHILE_BUSY: 从机在忙碌期间被禁用（发送、接收）。该位指示是否因 IC_ENABLE 寄存器第 0 位由 1 变为 0 而导致潜在或正在进行的从机操作被中止。当 CPU 在以下情况下向 IC_ENABLE 寄存器写入 0 时，该位被置位：</p> <p>(a) DW_apb_i2c 正在接收来自远程主控的从设备发送操作的地址字节；</p> <p>或者，</p> <p>(b) 来自远程主控的从设备接收操作的地址字节和数据字节。</p> <p>当读取值为1时，无论I2C地址是否与DW_apb_i2c（IC_SAR寄存器）中设定的从设备地址匹配，或者传输是否在IC_ENABLE设置为0后但尚未生效之前完成，DW_apb_i2c均视为已在I2C传输的任何阶段强制产生NACK。</p> <p>注意：如果远程I2C主控在DW_apb_i2c有机会响应NACK之前以STOP条件终止传输，且IC_ENABLE[0]已被设置为0，则该位也将被设置为1。</p> <p>当读取值为0时，DW_apb_i2c视为在主控活动期间被禁用，或当I2C总线处于空闲状态。</p> <p>注意：仅当 IC_EN（第 0 位）读取为 0 时，CPU 才可安全读取此位。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：从设备在空闲时被禁用		
	0x1 → 活动：从设备在激活时被禁用		
0	<p>IC_EN: ic_en 状态。该位始终反映输出端口的驱动值 ic_en。- 读取为1时，DW_apb_i2c 被视为处于启用状态。- 读取为0时，DW_apb_i2c 被视为完全不活动。注意：CPU 可在任何时间安全读取该位。当该位读取为0时，CPU 可安全读取 SLV_RX_DATA_LOST（位2）和 SLV_DISABLE_WHILE_BUSY（位1）。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 禁用：I2C 已禁用		
	0x1 → 启用：I2C 已启用		

I2C: IC_FS_SPKLEN 寄存器

偏移量：0xa0

描述

I2C SS、FS 或 FM+ 峰值抑制限制

该寄存器用于存储组件在 SS、FS 或 FM+ 模式下运行时，由脉冲抑制逻辑滤除的最长脉冲持续时间，单位为 ic_clk 时钟周期。相关的I2C要求是tSP（表

4) , 详见I2C总线规范。本寄存器必须设置为最小值1。

表490。
IC_FS_SPKLEN
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	IC_FS_SPKLEN: 必须在任何I2C总线事务发生前设置该寄存器, 以确保稳定运行。该寄存器以ic_clk周期为单位, 设定SCL或SDA线上最长尖峰的持续时间, 尖峰抑制逻辑将过滤该持续时间内的尖峰。仅当 I2C 接口被禁用时(对应 IC_ENABLE[0] 寄存器设为 0) , 才允许写入此寄存器。在其他时间写入无效。最小有效值为1; 硬件禁止写入小于该值的数值, 若尝试写入, 则自动设置为1。更多信息请参见“尖峰抑制”。	读写	0x07

I2C: IC_CLR_RESTART_DET寄存器

偏移: 0xa8

描述

清除 RESTART_DET 中断寄存器

表491。
IC_CLR_RESTART_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RESTART_DET: 读取该寄存器以清除IC_RAW_INTR_STAT寄存器中的RESTART_DET中断(第12位)。 复位值: 0x0	只读	0x0

I2C: IC_COMP_PARAM_1 寄存器

偏移: 0xf4

说明

组件参数寄存器 1

注意: 此寄存器未实现, 因此读取值恒为 0。若已实现, 该寄存器将为常量只读, 包含组件参数设置的编码信息。下表字段展示了相关参数的设定

表 492。
IC_COMP_PARAM_1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	TX_BUFFER_DEPTH: TX 缓冲区深度 = 16	只读	0x00
15:8	RX_BUFFER_DEPTH: RX 缓冲区深度 = 16	只读	0x00
7	ADD_ENCODED_PARAMS: 编码参数不可见	只读	0x0
6	HAS_DMA: DMA 握手信号已启用	只读	0x0
5	INTR_IO: 组合中断输出	只读	0x0
4	HC_COUNT_VALUES: 各模式的可编程计数值	只读	0x0
3:2	MAX_SPEED_MODE: 最大速度模式 = 快速模式	只读	0x0
1:0	APB_DATA_WIDTH: APB 数据总线宽度为 32 位	只读	0x0

I2C: IC_COMP_VERSION 寄存器

偏移: 0xf8

说明

I2C 组件版本寄存器

表 493。
IC_COMP_VERSION
寄存器

位	描述	类型	复位值
31:0	IC_COMP_VERSION	只读	0x3230312a

I2C: IC_COMP_TYPE 寄存器

偏移量: 0xfc

描述

I2C 组件类型寄存器

表 494。
IC_COMP_TYPE
寄存器

位	描述	类型	复位值
31:0	IC_COMP_TYPE : Designware 组件类型编号 = 0x44_57_01_40。该唯一十六进制值为常量，由两个 ASCII 字符 'DW' 及随后一个 16 位无符号数派生。	只读	0x44570140

4.4. SPI**ARM文档**

节录自 ARM PrimeCell 同步串行端口 (PL022) 技术参考手册，已获授权使用。

RP2040 配备两个相同的 SPI 控制器，均基于 ARM PrimeCell 同步串行端口 (SSP) (PL022) (修订版 r1p4)。注意，此接口与第 4.10 节中所述的 QSPI 接口不同。

每个控制器支持以下功能：

- 主模式或从模式
 - 兼容 Motorola SPI 接口
 - Texas Instruments 同步串行接口
 - National Semiconductor Microwire 接口
- 8 级深度的发送和接收 FIFO
- 用于服务 FIFO 或指示错误状态的中断生成
- 支持由 DMA 驱动
- 可编程时钟频率
- 可编程数据宽度，4 至 16 位

每个控制器可连接至多个 GPIO 引脚，详见第 2.19.2 节 GPIO 复用表 279。

GPIO 复用连接以 SPI 实例名称 `spi0_` 或 `spi1_` 为前缀，具体包括以下内容：

- 时钟 `sclk` (控制器作为主设备时连接至以下章节中的 SSPCLKOUT，作为从设备时连接至 SSPCLKIN)
- 低电平有效的片选或帧同步信号 `ss_n` (以下章节称为 SSPFSSOUT)
- 发送数据 `tx` (以下章节称为 SSPTXD，注意 nSSPOE 未连接至 `tx` 引脚，因此 SPI 控制器不会对输出数据进行三态控制)

- 接收数据 `rd` (以下章节称为SSPRXD)

SPI TX 引脚功能被设计为始终断言垫片输出使能信号，且不由 nSSPOE 进行驱动。当多个 SPI 从设备共享总线时，软件需切换输出使能信号。此操作可通过切换相关 `iobank0.ctrl` 寄存器的 oeover 字段，或通过切换 GPIO 功能实现。

SPI 使用 `clk_peri` 作为 SPI 定时的参考时钟，后续章节中称其为 SSPCLK。

`clk_sys` 用作总线时钟，后续章节中称其为 PCLK (另见第 2.15.1 节)。

4.4.1. 概述

PrimeCell SSP 是一主控或从属接口，用于与具备 Motorola SPI、National Semiconductor Microwire 或 Texas Instruments 同步串行接口的外围设备进行同步串行通信。

PrimeCell SSP 对从外围设备接收的数据执行串行向并行的转换。CPU 通过 AMBA APB 接口访问数据、控制及状态信息。发送和接收路径均采用内部 FIFO 存储器进行缓冲，使得在发送和接收模式下均可独立存储多达八个 16 位值。串行数据通过 SSPTXD 发送，并通过 SSPRXD 接收。

PrimeCell SSP 包含可编程的比特率时钟分频器和预分频器，根据输入时钟 SSPCLK 生成串行输出时钟 SSPCLKOUT。支持的比特率可达 2MHz 及以上，具体取决于 SSPCLK 的频率选择，最大比特率由外设设备决定。

您可以使用控制寄存器 SSPCR0 和 SSPCR1 来配置 PrimeCell SSP 的工作模式、帧格式及大小。

以下各个可屏蔽中断被触发：

- SSPTXINTR 请求服务发送缓冲区
- SSPRXINTR 请求服务接收缓冲区
- SSPRORINTR 表示接收 FIFO 出现溢出条件
- SSPRTINTR 表示在接收 FIFO 有数据存在时超时期限已到。

当任何单个中断被断言且未屏蔽时，会断言一个组合的单一中断。该中断连接至 RP2040 的处理器中断控制器。

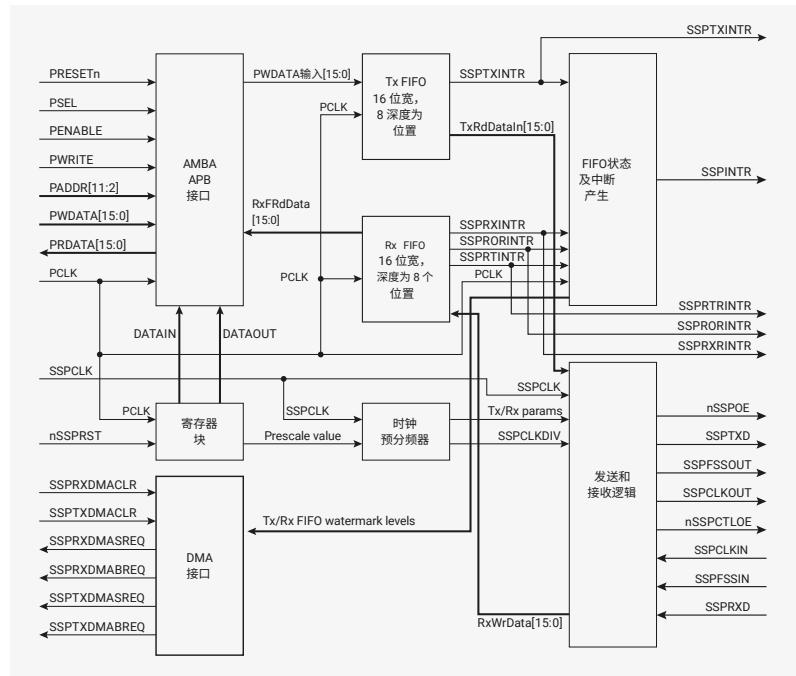
除了上述中断，还提供一组 DMA 信号，用于与 DMA 控制器接口。

根据所选工作模式，SSPFSSOUT 输出的功能为：

- 德州仪器同步串行帧格式的高电平有效帧同步输出。
- SPI 和 Microwire 的低电平有效从设备选择信号。

4.4.2. 功能描述

图87。PrimeCell
SSP模块示意图
为清晰起见，未
显示测试逻辑。



4.4.2.1 AMBA APB接口

AMBA APB接口生成用于访问状态和控制寄存器及传输与接收FIFO存储器的读写译码信号。

4.4.2.2 寄存器块

寄存器块存储通过AMBA APB接口写入或拟读取的数据。

4.4.2.3 时钟分频器

配置为主设备时，内部分频器由两个串联的可重载自由运行计数器组成，用以产生串行输出时钟SSPCLKOUT。

您可通过SSPCPSR寄存器设置时钟分频器，将SSPCLK按2至254之间的偶数因子分频。

因SSPCPSR寄存器未使用最低有效位，无法实现奇数分频，从而确保产生对称且占空比相等的时钟。详见SSPCPSR。

预分频器的输出通过编程SSPCRO控制寄存器，再次按1-256之间的因子分频，以产生最终的主控输出时钟SSPCLKOUT。

① 注意

图87中的PCLK和SSPCLK时钟输入分别连接至RP2040上的系统级时钟网`clk_sys`和`clk_peri`。默认情况下，`clk_peri`直接连接至系统时钟，但当系统时钟动态变化时，可解除连接以保持SPI频率恒定。有关RP2040时钟架构的概述，请参见图28。

4.4.2.4. 发送FIFO

通用发送FIFO为16位宽、深度为8个存储单元的缓冲器。通过AMBA APB总线写入的CPU数据

接口被存储于缓冲区，直至被发送逻辑读取。

配置为主机或从机时，平行数据在串行转换前写入发送FIFO，并分别通过SSPTXD引脚发送至所连接的从机或主机。

4.4.2.5. 接收 FIFO

通用接收 FIFO 是一个 16 位宽、深度为 8 个单元的内存缓冲区。从串行接口接收的数据存储在缓冲区中，直至通过 AMBA APB 接口由 CPU 读取。

当配置为主设备或从设备时，经过 SSPRXD 引脚接收的串行数据会先寄存，然后分别并行加载到附属的从设备或主设备接收 FIFO 中。

4.4.2.6. 发送与接收逻辑

当配置为主设备时，供给附属从设备的时钟信号来自 SSPCLK 经过预分频器处理后的降频信号。主设备的发送逻辑依次从其发送 FIFO 读取数据，并对其执行并行转串行转换。随后，同步于 SSPCLKOUT 的串行数据流及帧控制信号通过 SSP TXD 引脚输出至附属从设备。主设备的接收逻辑对输入的同步 SSPRXD 串行数据流执行串行转并行转换，将数值提取并存储至其接收 FIFO，以便通过 APB 接口后续读取。

当配置为从设备时，SSPCLKIN 时钟由连接的主设备提供，并用于定时传输和接收序列。从设备发送逻辑在主时钟控制下，依次从其发送 FIFO 读取数据，执行并行转串行转换，然后通过从设备 SSPTXD 引脚输出串行数据流及帧控制信号。从设备接收逻辑对输入的 SSPRXD 数据流执行串行转并行转换，提取并存入接收 FIFO，供后续通过 APB 接口读取。

4.4.2.7. 中断生成逻辑

PrimeCell SSP 产生四个可屏蔽的高电平有效独立中断。组合中断输出由各独立中断请求经逻辑或（OR）运算生成。

发送与接收动态数据流中断 SSPTXINTR 和 SSPRXINTR 与状态中断分开，以便根据 FIFO 触发级别读写数据。

4.4.2.8. DMA 接口

PrimeCell SSP 提供连接 DMA 控制器的接口，详见第 4.4.3.16 节。

4.4.2.9. 寄存器与逻辑的同步

PrimeCell SSP 支持 PCLK 和 SSPCLK 时钟的异步及同步操作。

同步寄存器和握手机制已实现，并始终处于激活状态。控制信号的同步在数据流的双向均已执行，具体为：

- 从 PCLK 域到 SSPCLK 域
- 从 SSPCLK 域到 PCLK 域。

4.4.3. 操作

4.4.3.1. 接口复位

PrimeCell SSP 通过全局复位信号 PRESETn 及模块专用复位信号 nSSPRST 进行复位。设备复位控制器异步置位 nSSPRST，并同步于 SSPCLK 时钟使其失效。

4.4.3.2. SSP 配置

复位后，PrimeCell SSP 逻辑被禁用，须在该状态下进行配置。必须编程控制寄存器 SSPCR0 和 SSPCR1，以配置外设为主机或从机，并使其在以下协议之一下工作：

- Motorola SPI
- Texas Instruments SSI
- National Semiconductor.

比特率由外部 SSPCLK 提供，需编程时钟预分频寄存器 SSPCPSR。

4.4.3.3. 启用 PrimeCell SSP 操作

您可以在 PrimeCell SSP 禁用时，通过写入最多八个16位值来预置发送 FIFO，或允许发送 FIFO 服务请求以中断 CPU。启用后，数据的传输或接收将通过传输引脚 SSPTXD 和接收引脚 SSPRXD 开始。

4.4.3.4. 时钟比率

PCLK 与 SSPCLK 频率比存在约束。SSPCLK 的频率必须小于或等于 PCLK 的频率，以确保从 SSPCLK 域到 PCLK 域的控制信号能在单帧周期内完成同步：

$$F_{SSPCLK} \leq F_{PCLK}$$

在从属模式下，来自外部主设备的 SSPCLKIN 信号经过双重同步后被延迟，以便检测边沿。检测 SSPCLKIN 上的边沿需三个 SSPCLK。SSPTXD 相对于主设备采样线路时的 SSPCLKIN 下降沿，其建立时间较短。

就 SSPCLKIN 而言，SSPRXD 的建立和保持时间必须更加保守，以确保在 SSPMS 内实际采样时值正确。为保证设备正常运行，SSPCLK 频率须至少为 SSPCLKIN 最大预期频率的 12 倍。

所选 SSPCLK 频率必须满足所需的位时钟速率范围。在从属模式下，SSPCLK 最低频率与 SSPCLKOUT 最大频率的比率为 12；在主模式下，该比率为 2。

例如，在 RP2040 上最大 SSPCLK (`clk_peri`) 频率为 133MHz 时，主模式的最大峰值比特率为 62.5Mbps。此速率通过将 SSPCPSR 寄存器设置为 2，且 SSPCR0 寄存器中的 SCR[7:0] 域设置为 0 实现。

在从属模式下，相同的最大 SSPCLK 频率 133MHz 可实现峰值比特率约为 $133 / 12 = 11.083$ Mbps。SSPCPSR 寄存器可设置为 12，SSPCR0 寄存器中的 SCR[7:0] 字段可设置为 0。同理，SSPCLK 最大频率与 SSPCLKOUT 最小频率的比值为 254×256 。

SSPCLK 的最低频率受以下不等式限制，且需同时满足：

$F_{SSPCLK}(min) \geq 2 \times F_{SSPCLKOUT}(max)$, 适用于主模式

$F_{SSPCLK}(min) \geq 12 \times F_{SSPCLKIN}(max)$, 适用于从模式。

SSPCLK 的最高频率受以下不等式限制，且需同时满足：

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKOUT}(min)$, 适用于主模式

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKIN}(min)$, 适用于从模式。

4.4.3.5. SSPCR0 控制寄存器的编程

SSPCR0 寄存器用于：

- 设置串行时钟速率
- 选择三种协议之一
- 选择数据字大小（如适用）

串行时钟速率 (SCR) 值结合 SSPCPSR 时钟预分频除数 CPSDVS, 用于从外部 SSPCLK 推导 PrimeCell SSP 的发送和接收比特率。

帧格式通过FRF位进行编程，数据字大小通过DSS位进行编程。

位相和极性，仅适用于Motorola SPI格式，通过SPH和SPO位进行编程。

4.4.3.6. SSPCR1控制寄存器的编程

SSPCR1寄存器用于：

- 选择主模式或从模式
- 启用环回测试功能
- 启用PrimeCell SSP外设。

要将PrimeCell SSP配置为主模式，请将SSPCR1寄存器中的主从选择位MS清零。该值为复位时的默认值。

将SSPCR1寄存器的MS位置1可将PrimeCell SSP配置为从模式。配置为从模式时，PrimeCell SSP的SSPTXD信号的启用或禁用由SSPCR1寄存器中从模式SSPTXD输出禁用位SOD控制。此功能在某些多从设备环境中有用，其中主设备可能进行并行广播。

要启用 PrimeCell SSP 的功能，请将同步串行端口使能位 (SSE) 设置为 1。

4.4.3.6.1. 比特率生成

串行比特率通过对输入时钟 SSPCLK 进行分频获得。时钟首先由一个在 2 至 254 范围内的偶数预分频值 CPSDVS 除，该值在 SSPCPSR 寄存器中配置。随后时钟再次被一个 1 至 256 范围内的值除，即 $1 + SCR$ ，其中 SCR 是在 SSPCR0 中编程设置的值。

以下公式定义了输出信号比特时钟 SSPCLKOUT 的频率：

$$F_{SSPCLKOUT} = \frac{F_{SSPCLK}}{CPSDVS \times (1 + SCR)}$$

例如，若 SSPCLK 为 125MHz，且 CPSDVS=2，则 SSPCLKOUT 的频率范围为 244kHz 至 62.5MHz。

4.4.3.7. 帧格式

每个数据帧长度介于 4 至 16 位之间，具体取决于配置的数据大小，传输时从最高有效位（MSB）开始。您可以选择以下基本帧类型：

- 德州仪器同步串行
- Motorola SPI
- 国家半导体Microwire。

对于所有格式，当PrimeCell SSP处于空闲状态时，串行时钟SSPCLKOUT保持非激活状态，仅在数据传输或接收期间以预定频率转换。SSPCLKOUT的空闲状态用于提供接收超时指示，当接收FIFO在超时期限后仍包含数据时触发该指示。

对于摩托罗拉SPI和国家半导体Microwire帧格式，串行帧信号SSPFSSOUT引脚为低有效，且在整个帧传输期间被拉低。

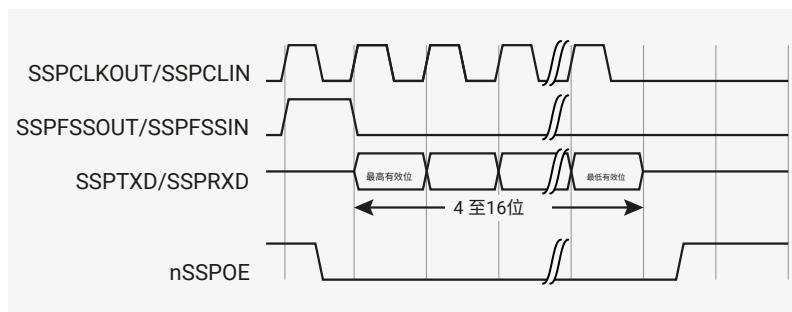
对于德州仪器同步串行帧格式，SSPFSSOUT引脚在每个帧传输开始前的上升沿触发一个串行时钟周期的脉冲。对于该帧格式，PrimeCell SSP及片外从设备均在SSPCLKOUT的上升沿驱动其输出数据，并在下降沿锁存对方设备的数据。

与另外两种帧格式的全双工传输不同，National Semiconductor Microwire 格式采用一种特殊的主从消息传递技术，工作于半双工模式。在该模式下，帧开始时会向芯片外的从设备发送一条8位控制消息。在此传输过程中，SSS 不接收任何输入数据。消息发送完成后，芯片外的从设备对其进行解码，并在最后一位8位控制消息发送结束后等待一个串行时钟周期，然后响应所请求数据。返回的数据长度可为4至16位，故总帧长度介于13至25位之间。

4.4.3.8. Texas Instruments同步串行帧格式

[图88展示了Texas Instruments单帧传输的同步串行帧格式。](#)

图88。德州仪器同步串行帧格式，单次传输

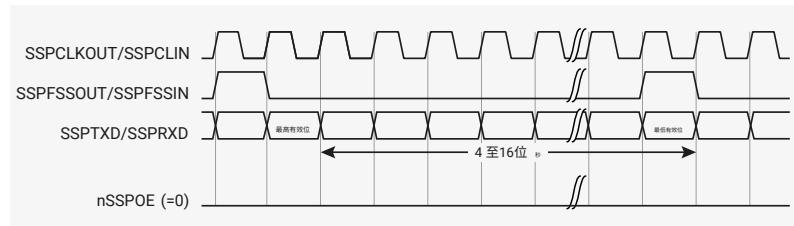


在此模式下，SSPCLKOUT 和 SSPFSSOUT 被强制拉低，当 PrimeCell SSP 空闲时，发送数据线 SSPTXD 处于三态。当发送 FIFO 底部条目包含数据时，SSPFSSOUT 会在一个 SSPCLKOUT 周期内被脉冲拉高。要传输的数值同时从发送 FIFO 转移至传输逻辑的串行移位寄存器。在下一次 SSPCLKOUT 上升沿时，4位至16位数据帧的最高有效位通过 SSPTXD 引脚移出。类似地，芯片外串行从设备将接收数据的最高有效位移入 SSPRXD 引脚。

PrimeCell SSP 与芯片外串行从设备均在每个 SSPCLKOUT 下降沿对各自串行移位寄存器中的数据位进行时钟驱动。接收数据在最低有效位锁存后，于 PCLK 首个上升沿从串行移位寄存器转移至接收 FIFO。

[图89展示了德州仪器连续传输的同步串行帧格式。](#)

图89。德州仪器同步串行帧格式，连续传输



4.4.3.9. Motorola SPI帧格式

Motorola SPI接口为四线接口，其中SSPFSSOUT信号用作从设备选择信号。Motorola SPI格式的主要特征是可通过SSPCRO控制寄存器中的SPO和SPH位来编程设置SSPCLKOUT信号的非活动状态和相位。

4.4.3.9.1. SPO，时钟极性

当SPO时钟极性控制位为低时，SSPCLKOUT引脚保持稳定的低电平；当SPO时钟极性控制位为高时，数据未传输期间，SSPCLKOUT引脚保持稳定的高电平。

4.4.3.9.2. SPH，时钟相位

SPH控制位用于选择捕获数据的时钟边沿，并允许其状态改变。该控制位对第一个传输的数据位影响最大，决定是否允许在第一个数据捕获边沿之前发生时钟跳变。

当SPH相位控制位为低电平时，数据在第一个时钟跳变边沿被捕获。

当SPH时钟相位控制位为高电平时，数据在第二个时钟跳变边沿被捕获。

4.4.3.10. Motorola SPI 格式，SPO=0，SPH=0

图90与图91展示了SPO=0，SPH=0条件下Motorola SPI帧格式的连续传输信号序列。图90展示了SPO=0，SPH=0条件下的单次传输信号序列。

图90。Motorola SPI帧格式，单次传输， $SPO=0$, $SPH=0$

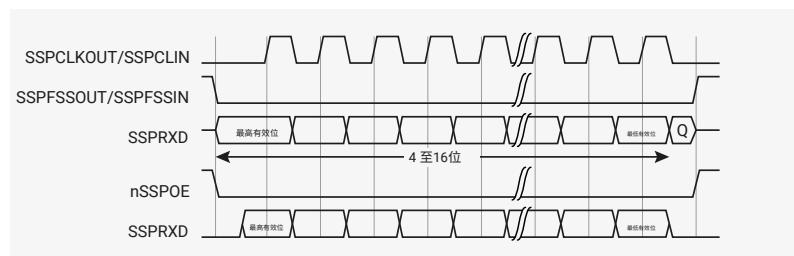
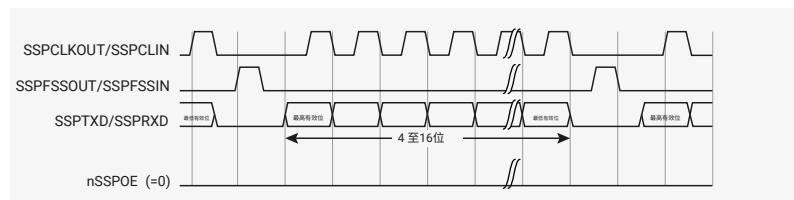


图91展示了SPO=0且SPH=0的Motorola SPI帧格式下的连续传输信号序列。

图91。Motorola SPI帧格式，单次传输， $SPO=0$ 且 $SPH=0$



在此配置下，空闲期间：

- SSPCLKOUT信号被强制保持低电平

- SSPFSSOUT信号被强制保持高电平
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

若启用PrimeCell SSP且传输FIFO内含有效数据，传输开始由主设备信号SSPFSSOUT拉低表示，此时从设备数据被使能至主设备的SSPRXD输入线。nSSPOE 线路被驱动为低电平，从而使主设备 SSPTXD 输出引脚生效。

半个 SSPCLKOUT 周期后，有效的主设备数据被传输至 SSPTXD 引脚。既然主设备和从设备的数据均已设定，SSPCLKOUT 主时钟引脚将在额外半个 SSPCLKOUT 周期后变为高电平。

数据现于 SSPCLKOUT 信号的上升沿被捕获，并于下降沿被传播。

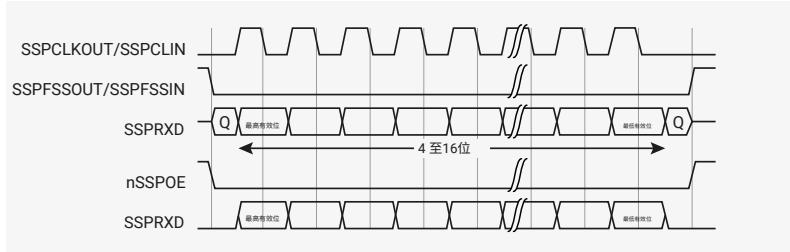
在单字传输情况下，所有数据字比特传输完成后，SSPFSSOUT 线将在最后一个比特捕获后一个 SSPCLKOUT 周期内恢复至其空闲高电平状态。

然而，在连续背靠背的传输中，必须在每个数据字传输之间对 SSPFSSOUT 信号施加高电平脉冲。这是因为从设备选择引脚会冻结串行外设寄存器中的数据，若 SPH 位为逻辑零，则不允许数据被更改。因此，主设备必须在每次数据传输之间拉高从设备的 SSPFSIN 引脚，以启用串行外设数据写入。连续传输结束后，最后一位被捕获后一个 SSPCLKOUT 周期，SSPFSSOUT 引脚返回其空闲状态。

4.4.3.11. SPO=0, SPH=1 的 Motorola SPI 格式

图 92 显示了 SPO=0, SPH=1 的 Motorola SPI 格式传输信号序列，涵盖单次及连续传输。

图 92。SPO=0 且
SPH=1 的 Motorola S
PI 帧格式，单次
及连续传输



在此配置下，空闲期间：

- SSPCLKOUT信号被强制保持低电平
- SSPFSSOUT 信号被强制拉高。
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

如果 PrimeCell SSP 启用且传输 FIFO 中存在有效数据，传输起始由 SSPFSSOUT 主控信号拉低表示。nSSPOE 线被拉低，启用主控 SSPTXD 输出引脚。在额外一个半 SSPCLKOUT 周期后，主从设备的有效数据同步使能于各自传输线。同时，SSPCLKOUT 在上升沿触发时被使能。

数据随后在 SSPCLKOUT 信号的下降沿被采集，并在上升沿传输。

对于单字传输，在所有位传输完成后，SSPFSSOUT 线将在最后一位被采集后的一个 SSPCLKOUT 周期内回到其空闲的高电平 (HIGH) 状态。对于连续的背靠背传输，SSPFSSOUT 引脚在连续数据字之间保持低电平 (LOW)，其终止方式与单字传输相同。

4.4.3.12. Motorola SPI 格式，SPO=1，SPH=0

图93和图94展示了 SPO=1、SPH=0 的 Motorola SPI 格式下的单次及连续传输信号序列。

图93展示了 SPO=1、SPH=0 的 Motorola SPI 格式下的单次传输信号序列。

图93。Motorola S PI帧格式，单次传输，SPO=1且SPH=0

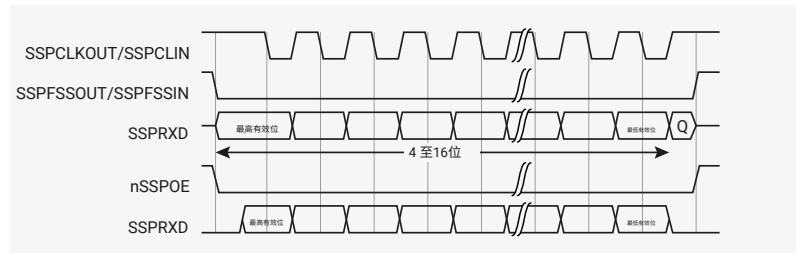
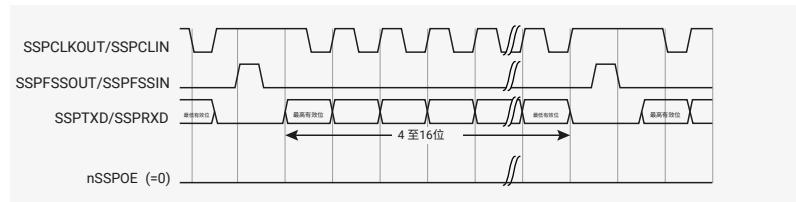


图94展示了SPO=1、SPH=0的Motorola SPI格式下的连续传输信号序列。

i 注意

图93中信号Q未定义。

图94。Motorola S PI帧格式，连续传输，SPO=1且SPH=0。



在此配置下，空闲期间：

- SSPCLKOUT信号被强制置为高电平。
- SSPFSSOUT信号被强制保持高电平
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

若启用PrimeCell SSP且发送FIFO中存在有效数据，则传输起始由SSPFSSOUT主控信号拉低表示，导致从设备数据立即传输至主设备的SSPRXD线上。nSSPOE 线路被驱动为低电平，从而使主设备 SSPTXD 输出引脚生效。

半个周期后，有效的主设备数据传输至SSPTXD线上。主从设备数据均设置完毕后，SSPCLKOUT主时钟引脚在额外半个SSPCLKOUT周期后变为低电平。这表示数据在SSPCLKOUT信号的下降沿被捕获，并在上升沿传播。

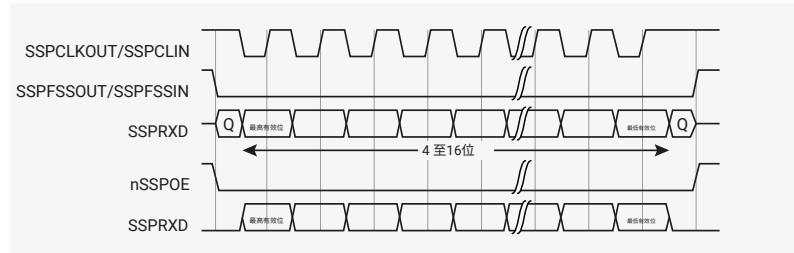
单字传输时，当数据字所有位传输完成后，SSPFSSOUT线于最后一位捕获后一个SSPCLKOUT周期内恢复至其空闲高电平状态。

然而，在连续背靠背的传输中，必须在每个数据字传输之间对 SSPFSSOUT 信号施加高电平脉冲。这是因为从设备选择引脚会冻结串行外设寄存器中的数据，若 SPH 位为逻辑零，则不允许数据被更改。因此，主设备必须在每次数据传输之间拉高从设备的 SSPFSSIN 引脚，以启用串行外设数据写入。连续传输结束后，最后一位被捕获后一个 SSPCLKOUT 周期，SSPFSSOUT 引脚返回其空闲状态。

4.4.3.13. SPO=1, SPH=1 的 Motorola SPI 格式

图95展示了 SPO=1, SPH=1 的 Motorola SPI 格式的传输信号序列，涵盖单次和连续传输。

图95。SPO=1 和
SPH=1 的 Motorola S
PI 帧格式，支持
单次及连续传输



① 注意

图95中，Q 为未定义信号。

在此配置下，空闲期间：

- SSPCLKOUT 信号被强制置为高电平。
- SSPFSSOUT 信号被强制保持高电平
- 传输数据线 SSPTXD 被任意强制保持低电平
- nSSPOE 引脚使能信号被强制保持高电平（注意，该信号在 RP2040 中未连接至引脚）
- 当 PrimeCell SSP 配置为主设备时，nSSPCTLOE 线被拉低，启用 SSPCLKOUT 引脚，该使能信号为低有效
- 当 PrimeCell SSP 配置为从设备时，nSSPCTLOE 线被拉高，禁用 SSPCLKOUT 引脚，该使能信号为低有效。

若启用 PrimeCell SSP 且发送 FIFO 中存在有效数据，则传输起始由 SSPFSSOUT 主控信号被拉低表示。nSSPOE 线拉低，主设备 SSPTXD 输出端口被使能。再经过半个 SSPCLKOUT 周期后，主从设备的数据均被允许输出至各自传输线。同时，SSPCLKOUT 信号以下降沿转换方式被使能。数据随后在 SSPCLKOUT 信号的上升沿被采集，并于下降沿发出。

在所有位传输完成后，对于单字传输，SSPFSSOUT 线路将在最后一位被捕获后一个 SSPCLKOUT 周期恢复至空闲的高电平状态。

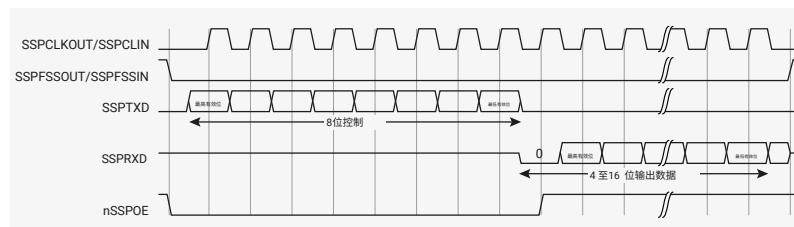
对于连续背靠背传输，SSPFSSOUT 引脚保持活动的低电平状态，直到最后一字的最后一位被捕获，然后如前文所述返回至空闲状态。

对于连续背靠背传输，SSPFSSOUT 引脚在连续数据字之间保持低电平，终止方式与单字传输相同。

4.4.3.14. National Semiconductor Microwire 帧格式

图96展示了 National Semiconductor Microwire 的单帧格式。图97展示了连续帧传输时的相同格式。

图96。Microwire
帧格式，单次
传输



Microwire格式与SPI格式极为相似，不同之处在于传输采用半双工而非全双工，使用主从消息传递技术。每次串行传输均始于一个由PrimeCell SSP发送至片外从设备的8位控制字。在该传输过程中，PrimeCell SSP不会接收任何输入数据。消息发送完成后，片外从设备对其进行解码，并在最后一位8位控制消息发送完成后等待一个串行时钟周期，然后返回所需数据。返回的数据长度为4至16位，使得总帧长度在13至25位之间。

在此配置下，空闲期间：

- SSPCLKOUT被强制为低电平
- SSPFSSOUT被强制为高电平
- 发送数据线SSPTXD被任意强制为低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）

通过向发送FIFO写入控制字节来触发传输。SSPFSSOUT的下降沿促使发送FIFO底部的数据被传输至发送逻辑的串行移位寄存器，且8位控制帧的最高位（MSB）被移出至SSPTXD引脚。在整个帧传输期间，SSPFSSOUT保持低电平。在此传输期间，SSPRXD引脚保持三态（高阻）状态。

片外串行从设备在每个SSPCLKOUT上升沿将每个控制位锁存至其串行移位寄存器中。

在从设备锁存最后一位后，控制字节将在一个时钟等待周期内被译码，从设备随后通过传输数据响应PrimeCell SSP。每个位于SSPCLKOUT下降沿驱动至SSPRXD线，PrimeCell SSP则于SSPCLKOUT上升沿锁存每个位。帧结束时，对于单次传输，SSPFSSOUT信号在接收串行移位寄存器锁存最后一位后一个时钟周期被拉高，促使数据传输至接收FIFO。

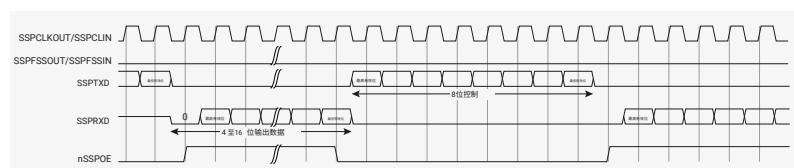
注意

片外设备可在接收移位器锁存最低有效位后的SSPCLKOUT下降沿，或SSPFSSOUT引脚变为高电平时，将接收线置于三态（高阻）状态。

对于连续传输，数据传输的开始和结束方式与单次传输相同。然而，SSPFSSOUT线持续断言，保持低电平，且数据连续传输。下一帧的控制字节紧随当前帧接收数据的最低有效位之后。每个接收值均在SSPCLKOUT下降沿由接收移位器传输，前提是该帧的最低有效位已锁存至PrimeCell SSP。

图97展示了在连续传输多帧时，National Semiconductor Microwire 的帧格式。

图97。Microwire
帧格式，
连续传输



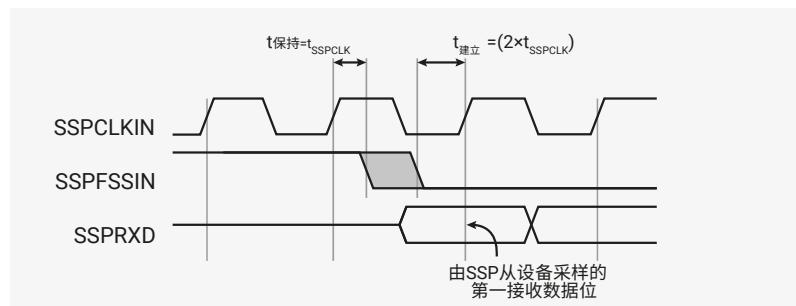
在Microwire模式下，PrimeCell SSP从属设备在SSPFSSIN拉低后，于SSPCLKIN上升沿采样接收数据的第一个位。驱动自由运行SSPCLKIN的主设备必须确保SSPFSSIN信号相对于SSPCLKIN上升沿具备充分的建立时间和保持时间裕量。

图98显示了这些建立时间和保持时间的要求。

关于PrimeCell SSP从设备采样接收数据的第一位时所对应的SSPCLKIN上升沿，SSPFSSIN必须具有至少两倍于PrimeCell SSP运行时所用SSPCLK周期的建立时间。

关于该上升沿之前的SSPCLKIN上升沿，SSPFSSIN必须保持至少一个SSPCLK周期的保持时间。

图98。Microwire
帧格式，SS
PFSSIN输入的建立
和保持时间要求



4.4.3.15. 主从设备配置示例

图99、图100和图101展示了当PrimeCell SSP（PL022）外设配置为主设备或从设备时，如何连接其他同步串行外设。

注意

SSP（PL022）不支持系统中主从角色的动态切换。每个实例均被配置并连接为主设备或从设备。

图99. PrimeCell
SSP主设备连接至
PL022从设备

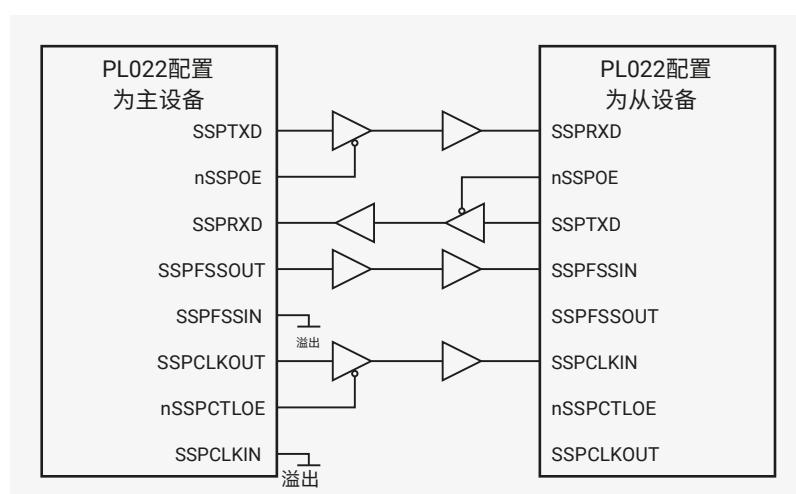


图100展示了配置为主设备的PrimeCell SSP（PL022）如何与摩托罗拉SPI从设备接口。SPI从设备选择（SS）信号被永久拉低，用于将其配置为从设备。与上述操作类似，主设备可通过主设备PrimeCell SSP的SSPTXD线向从设备广播。作为响应，从设备将其SPI MISO端口驱动至主设备的SSPRXD线上。

图 100。PrimeCell
SSP 主设备与 SPI 从
设备的连接

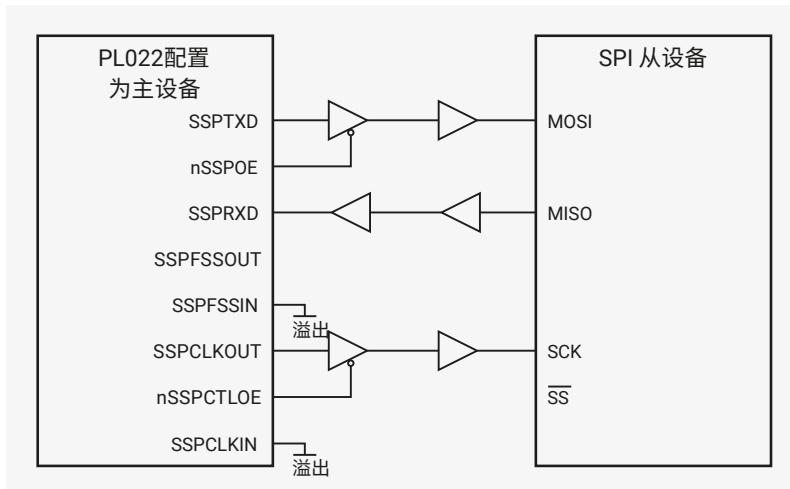
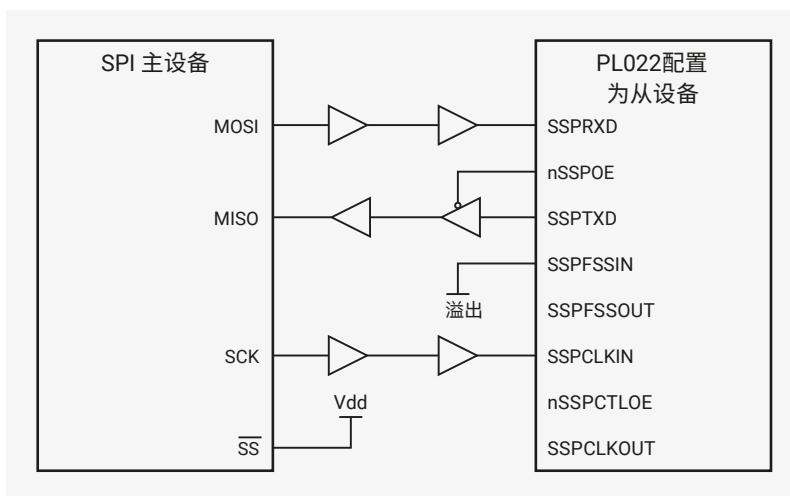


图 101 显示了一台配置为主设备的 Motorola SPI，连接至配置为从设备的 PrimeCell SSP（PL022）实例。在此情况下，从设备的选择信号（SS）被永久拉高，以将其配置为主设备。主设备可通过主设备 SPI MOSI 线向从设备广播，作为响应，从设备将其 nSSPOE 信号拉低，从而使其 SSPTXD 数据能够输出至主设备的 MISO 线上。

图 101。SPI 主控
连接至 PrimeCell
SSP 从属设备



4.4.3.16. PrimeCell DMA 接口

PrimeCell SSP 提供连接至 DMA 控制器的接口。PrimeCell SSP 的 DMA 控制寄存器 SSPDMACR 用于控制 PrimeCell SSP 的 DMA 操作。

DMA 接口包括以下接收信号：

SSPRXDMASREQ

单字符 DMA 传输请求，由 SSP 断言。当接收 FIFO 中至少有一个字符时，该信号被断言。

SSPRXDMABREQ

突发 DMA 传输请求，由 SSP 断言。当接收 FIFO 中包含四个或更多字符时，该信号被断言。

SSPRXDMACLR

DMA 请求清除，由 DMA 控制器断言，用以清除接收请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

DMA 接口包括以下发送信号：

SSPTXDMASREQ

单字符 DMA 传输请求，由 SSP 断言。当发送 FIFO 中至少有一个空位时，该信号被断言。

SSPTXDMABREQ

突发 DMA 传输请求，由 SSP 断言。当发送 FIFO 中包含四个或更少字符时，该信号被断言。

SSPTXDMACLR

DMA 请求清除信号，由 DMA 控制器断言，用于清除传输请求信号。若请求 DMA 突发传输，则在传输突发的最后一笔数据时断言清除信号。

突发传输请求信号与单次传输请求信号并非互斥。二者可以同时断言。例如，当接收 FIFO 中的数据量超过四的水位线时，突发传输请求和单次传输请求会被断言。当接收 FIFO 中剩余数据量低于水位线时，仅断言单次传输请求。该设置适用于流中剩余待接收字符数少于一个突发长度的情况。

例如，若需接收 19 个字符，DMA 控制器先传输四个 4 字符的突发，随后进行 3 次单次传输以完成数据流。

注意

对于剩余的 3 个字符，PrimeCell SSP 不会断言突发请求信号。

每个请求信号将持续断言，直到相应的 DMA 清除信号被断言。在请求清除信号撤销后，请求信号可再次变为有效，具体取决于前述章节所述的条件。如果 PrimeCell SSP 被禁用，或者 DMA 使能信号被清除，则所有请求信号均被撤销。

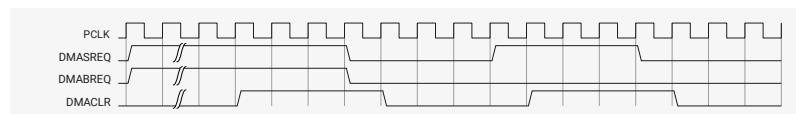
[表 495 显示了 DMABREQ 对于发送和接收 FIFO 的触发点。](#)

表 495。发送与接收 FIFO 的 DMA 触发点

突发长度		
水位线	发送端，空闲位置数量	接收端，已占用位置数量
1/2	4	4

[图 102 展示了在适用的 DMA 清除信号下，单次传输请求和突发传输请求的时序图。所有信号均与 PCLK 同步。](#)

图 102。DMA 传输波形



4.4.4. 寄存器列表

SPI0 和 SPI1 寄存器的基址分别为 [0x4003c000](#) 和 [0x40040000](#)（在 SDK 中定义为 SPI0_BASE 和 SPI1_BASE）。

表 496. SPI 寄存器列表

偏移量	名称	说明
0x000	SSPCR0	控制寄存器 0 SSPCR0，详见第 3-4 页
0x004	SSPCR1	控制寄存器 1 SSPCR1，详见第 3-5 页
0x008	SSPDR	数据寄存器 SSPDR，详见第 3-6 页
0x00c	SSPSR	状态寄存器 SSPSR，详见第 3-7 页
0x010	SSPCPSR	时钟预分频寄存器 SSPCPSR，详见第 3-8 页

偏移量	名称	说明
0x014	SSPIMSC	中断屏蔽设置/清除寄存器 SSPIMSC，详见第3-9页
0x018	SSPRIS	原始中断状态寄存器 SSPRIS，详见第3-10页
0x01c	SSPMIS	屏蔽中断状态寄存器 SSPMIS，详见第3-11页
0x020	SSPICR	中断清除寄存器，SSPICR，见第3-11页
0x024	SSPDMACR	DMA控制寄存器，SSPDMACR，见第3-12页
0xfe0	SSPPERIPHIDO	外设标识寄存器，SSPPERIPHIDO-3，见第3-13页
0xfe4	SSPPERIPHID1	外设标识寄存器，SSPPERIPHID1-3，见第3-13页
0xfe8	SSPPERIPHID2	外设标识寄存器，SSPPERIPHID2-3，见第3-13页
0xfec	SSPPERIPHID3	外设标识寄存器，SSPPERIPHID3-3，见第3-13页
0xff0	SSPPCELLID0	PrimeCell标识寄存器，SSPPCELLID0-3，见第3-16页
0xff4	SSPPCELLID1	PrimeCell标识寄存器，SSPPCELLID1-3，见第3-16页
0xff8	SSPPCELLID2	PrimeCell标识寄存器，SSPPCELLID2-3，见第3-16页
0ffc	SSPPCELLID3	PrimeCell标识寄存器，SSPPCELLID3-3，见第3-16页

SPI: SSPCR0 寄存器

偏移：0x000

描述

控制寄存器0 SSPCR0，详见第3-4页

表 497. SSPCR0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:8	SCR : 串行时钟速率。SCR值用于生成PrimeCell SSP的发送与接收比特率。比特率计算公式为： $F_{SSPCLK} \div (CPSDVSR \times (1+SCR))$ ，其中CPSDVSR为通过SSPCPSR寄存器配置的2至254之间的偶数，SCR则为0至255之间的值。	读写	0x00
7	SPH : SSPCLKOUT 相位，仅适用于 Motorola SPI 帧格式。请参见第 2-10 页的 Motorola SPI 帧格式说明。	读写	0x0
6	SPO : SSPCLKOUT 极性，仅适用于 Motorola SPI 帧格式。请参见第 2-10 页的 Motorola SPI 帧格式说明。	读写	0x0
5:4	FRF : 帧格式：00 Motorola SPI 帧格式；01 TI 同步串行帧格式；10 National Semiconductor Microwire 帧格式；11 保留，未定义操作。	读写	0x0
3:0	DSS : 数据位选择：0000 保留，未定义操作；0001 保留，未定义操作；0010 保留，未定义操作；0011 4 位数据。0100 5 位数据；0101 6 位数据；0110 7 位数据；0111 8 位数据；1000 9 位数据；1001 10 位数据；1010 11 位数据；1011 12 位数据；1100 13 位数据。1101 14 位数据；1110 15 位数据；1111 16 位数据。	读写	0x0

SPI: SSPCR1 寄存器

偏移：0x004

描述

控制寄存器1 SSPCR1，详见第3-5页

表 498. SSPCR1
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	SOD : 从机模式输出禁用。此位仅在从机模式 (MS=1) 下有效。在多从机系统中, PrimeCell SSP 主设备可以向系统中所有从设备广播消息, 同时确保只有一个从设备驱动其串行输出线路的数据。在此类系统中, 多个从设备的 RXD 线路可能被并联连接。为保证在此类系统中正常工作, 如 PrimeCell SSP 从设备不应驱动 SSPTXD 线路, 则可设置 SOD 位: 0 表示 SSP 在从机模式下可驱动 SSPTXD 输出; 1 表示 SSP 在从机模式下不得驱动 SSPTXD 输出。	读写	0x0
2	MS : 主从模式选择。该位仅可在 PrimeCell SSP 禁用时修改 (SSE=0) : 0 表示设备配置为主设备, 默认值; 1 表示设备配置为从设备。	读写	0x0
1	SSE : 同步串行端口使能: 0 表示禁用 SSP 操作。1 表示启用 SSP 操作。	读写	0x0
0	LBM : 环回模式: 0 启用正常串行端口操作。1 传输串行移位寄存器的输出在内部连接至接收串行移位寄存器的输入。	读写	0x0

SPI: SSPDR 寄存器

偏移: 0x008

说明

数据寄存器 SSPDR, 详见第3-6页

表 499. SSPDR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	DATA : 传输/接收 FIFO: 读取接收 FIFO, 写入传输 FIFO。当 PrimeCell SSP 被配置为小于 16 位的数据大小时, 数据必须右对齐。传输逻辑将忽略高位未使用的位。 接收逻辑会自动进行右对齐。	RWF	-

SPI: SSPSR 寄存器

偏移: 0x00c

说明

状态寄存器 SSPSR, 详见第3-7页

表 500. SSPSR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4	BSY : PrimeCell SSP 忙标志, RO: 0 表示 SSP 空闲。1 表示 SSP 正在发送和/或接收一帧, 或传输 FIFO 非空。	只读	0x0
3	RFF : 接收FIFO满, RO: 0 表示接收FIFO未满, 1 表示接收FIFO已满。	只读	0x0
2	RNE : 接收FIFO非空, RO: 0 表示接收FIFO为空, 1 表示接收FIFO非空。	只读	0x0
1	TNF : 发送FIFO未满, RO: 0 表示发送FIFO已满, 1 表示发送FIFO未满。	只读	0x1
0	TFE : 发送FIFO空, RO: 0 表示发送FIFO非空, 1 表示发送FIFO为空。	只读	0x1

SPI: SSPCPSR 寄存器

偏移: 0x010

描述

时钟预分频寄存器 SSPCPSR，详见第3-8页

表 501. SSPCPSR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	CPSDVS : 时钟预分频除数。必须为2至254之间的偶数，具体取决于SS PCLK的频率。读取时最低有效位始终返回零。	读写	0x00

SPI: SSPIMSC 寄存器

偏移: 0x014

描述

中断屏蔽设置/清除寄存器 SSPIMSC，详见第3-9页

表 502. SSPIMSC
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXIM : 发送FIFO中断屏蔽：0 表示发送FIFO半空或更少状态中断被屏蔽，1 表示发送FIFO半空或更少状态中断未被屏蔽。	读写	0x0
2	RXIM : 接收FIFO中断屏蔽：0 接收FIFO半满或以下条件中断被屏蔽。1 接收FIFO半满或以下条件中断未被屏蔽。	读写	0x0
1	RTIM : 接收超时中断屏蔽：0 接收FIFO非空且超时期间未读取的中断被屏蔽。1 接收FIFO非空且超时期间未读取的中断未被屏蔽。	读写	0x0
0	RORIM : 接收溢出中断屏蔽：0 接收FIFO满时写入的中断被屏蔽。1 接收FIFO满时写入的中断未被屏蔽。	读写	0x0

SPI: SSPRIS寄存器

偏移量: 0x018

描述

原始中断状态寄存器 SSPRIS，详见第3-10页

表 503. SSPRIS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXRIS : 反映SSPTXINTR中断的原始状态（未屏蔽）	只读	0x1
2	RXRIS : 反映SSPRXINTR中断的原始状态（未屏蔽）	只读	0x0
1	RTRIS : 反映SSPRTINTR中断的原始状态（未屏蔽）	只读	0x0

位	描述	类型	复位值
0	RORRIS : 提供SSPRORINTR中断在屏蔽前的原始中断状态	只读	0x0

SPI: SSPMIS寄存器

偏移: 0x01c

说明

屏蔽中断状态寄存器 SSPMIS, 详见第3-11页

表504. SSPMIS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXMIS : 提供SSPTXINTR中断在屏蔽后的发送FIFO屏蔽中断状态 SSPTXINTR中断	只读	0x0
2	RXMIS : 提供SSPRXINTR中断在屏蔽后的接收FIFO屏蔽中断状态	只读	0x0
1	RTMIS : 提供SSPRTINTR中断在屏蔽后的接收超时屏蔽中断状态	只读	0x0
0	RORMIS : 提供SSPRORINTR中断在屏蔽后的接收溢出屏蔽中断状态	只读	0x0

SPI: SSPICR寄存器

偏移: 0x020

说明

中断清除寄存器, SSPICR, 见第3-11页

表505. SSPICR
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	RTIC : 清除SSPRTINTR中断	WC	0x0
0	RORIC : 清除SSPRORINTR中断	WC	0x0

SPI: SSPDMACR寄存器

偏移: 0x024

描述

DMA控制寄存器, SSPDMACR, 见第3-12页

表 506. SSPDMACR
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TXDMAE : 发送 DMA 启用。若该位设置为 1，则启用发送 FIFO 的 DMA。	读写	0x0
0	RXDMAE : 接收 DMA 启用。若该位设置为 1，则启用接收 FIFO 的 DMA。	读写	0x0

SPI: SSPPERIPHIDO 寄存器

偏移量: 0xfe0

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 507
SSPPERIPHID
0 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	PARTNUMBER0 : 此位读取值为 0x22	只读	0x22

SPI: SSPPERIPHID1 寄存器

偏移: 0xfe4

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 508
SSPPERIPHID
1 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	DESIGNERO : 这些位的读出值为0x1	只读	0x1
3:0	PARTNUMBER1 : 这些位读取值为 0x0	只读	0x0

SPI: SSPPERIPHID2 寄存器

偏移: 0xfe8

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 509
SSPPERIPHID
2 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	REVISION : 此位返回外设修订版本	只读	0x3
3:0	DESIGNER1 : 这些位读取值为 0x4	只读	0x4

SPI: SSPPERIPHID3 寄存器

偏移: 0fec

说明

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 510
SSPPERIPHID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	配置 : 这些位读取值为 0x00	只读	0x00

SPI: SSPPCELLID0 寄存器

偏移量: 0xff0

描述

PrimeCell标识寄存器，SSPPCellID0-3，见第3-16页

表 511
SSPPCELLID0 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID0 : 这些位读取值为 0x0D	只读	0x0d

SPI: SSPPCELLID1 寄存器

偏移: 0xff4

说明

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 512
SSPPCELLID1 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID1 : 这些位读取值为 0xF0	只读	0xf0

SPI: SSPPCELLID2 寄存器

偏移: 0xff8

说明

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 513
SSPPCELLID2 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID2 : 这些位读取值为 0x05	只读	0x05

SPI: SSPPCELLID3 寄存器

偏移: 0ffc

描述

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 514。
SSPPCELLID3 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID3 : 这些位读回值为 0xB1	只读	0xb1

4.5. PWM

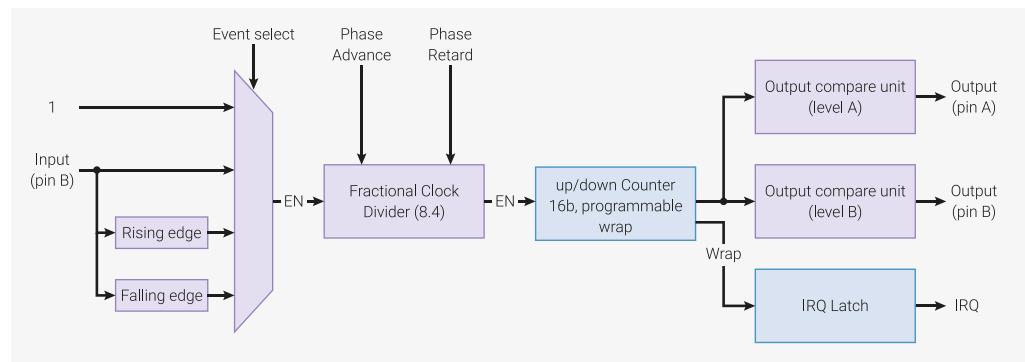
4.5.1. 概述

脉冲宽度调制（PWM）是一种通过数字信号提供平滑变化平均电压的方案。该方案通过在固定间隔内施加宽度受控的正脉冲实现。高电平持续时间占总周期的比例称为占空比。该方法可用于模拟输出的近似，或控制开关模式电源电子设备。

RP2040 的 PWM 模块包含8个功能相同的切片。每个切片可驱动两个 PWM 输出信号，或测量输入信号的频率或占空比，故总共可控制多达16个 PWM 输出。所有30个 GPIO 引脚均可由 PWM 模块驱动。

图103。一个单个 PWM 切片。一个16位计数器从0计数至设定值，然后根据 PWM 模式环绕回零，或反向计数。A 和 B 输出根据当前计数值及预设的 A 和 B 阀值在高低电平之间切换。计数器基于多个事件递增：它

可以自由运行，或通过 B 引脚输入信号的电平或边沿进行门控。分数时钟除频器降低整体计数速率，以实现更精细的输出频率控制。



每个 PWM 单元配备以下功能：

- 16 位计数器
- 8.4 分数时钟除频器
- 两个独立输出通道，占空比范围为 0% 至 100% 包含端点
- 双斜率及尾部边沿调制
- 用于频率测量的边沿敏感输入模式
- 用于占空比测量的电平敏感输入模式
- 可配置的计数器循环值
 - 循环值和电平寄存器采用双缓冲设计，可在 PWM 运行时无竞争地更新
- 计数器循环时产生中断请求和 DMA 请求
- 相位可在运行时精确提前或延迟（以单计数步进）

可通过单一全局控制寄存器同时启用或禁用所有单元。切片随后以完美同步的方式运行，以便通过多个切片的输出切换更复杂的电源电路。

4.5.2. 程序员模型

RP2040的所有30个GPIO引脚均可用于PWM：

表515。RP2040上PWM通道与GPIO引脚的映射。
该信息亦显示于主GPIO功能表
(表279) 中。

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM通道	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM通道	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

- 这16个PWM通道（8个双通道切片）分布在GPIO0至GPIO15，引脚顺序为PWM0 A、PWM0 B、PWM1 A...
- 此规律重复应用于GPIO16至GPIO29。GPIO16为PWM0 A，GPIO17为PWM0 B，依此类推，至GPIO29的PWM6 B
- 同一PWM信号可被选择输出至两个GPIO引脚；相同信号将同时出现在各GPIO引脚上。
- 若PWM B引脚用作输入且被多个GPIO引脚选中，则PWM切片会接收这些GPIO输入的逻辑或信号

4.5.2.1. 脉冲宽度调制

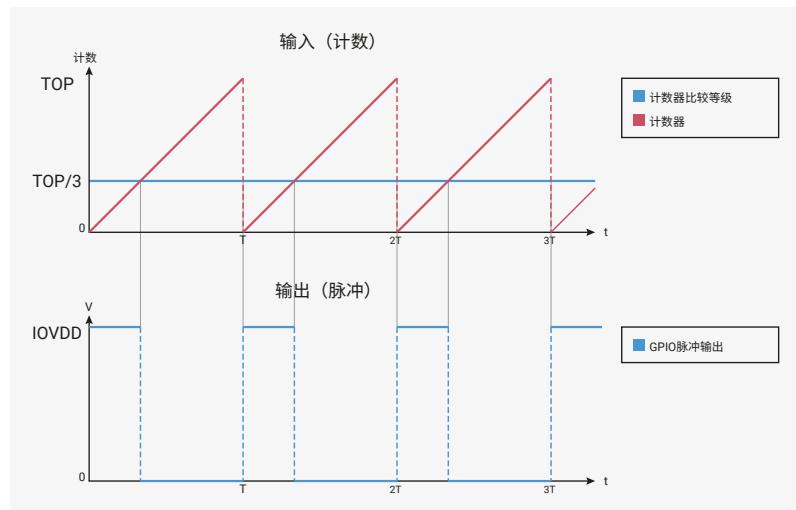
PWM硬件通过持续比较输入值与自由运行计数器实现其功能。该过程生成切换输出，其中高电平输出时间的长短与输入值成比例。高电平信号维持的时间比例即为信号的占空比。

计数周期由 **TOP** 寄存器控制，最大周期为 65536 个计数周期，因计数器和 **TOP** 均为 16 位。输入数值通过 **CC** 寄存器进行配置。

图104。计数器反复从0计数至TOP，形成锯齿波形。计数器数值会持续与某一输入值比较。当输入值大于计数器值时，输出为高电平。

否则，输出为低电平。输出周期T由计数器的TOP值及计数速度决定。

作为I/O电源电压的比例，平均输出电压为输入值除以计数周期($TOP + 1$)



本示例展示了如何在RP2040的某个PWM切片上配置计数周期及A、B计数比较电平。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pwm/hello_pwm/hello_pwm.c 第14至29行

```

14 // 告诉GPIO 0和1它们被分配为PWM功能
15 gpio_set_function(0, GPIO_FUNC_PWM);
16 gpio_set_function(1, GPIO_FUNC_PWM);
17
18 // 查询连接到GPIO 0的PWM切片（即切片0）
19 uint slice_num = pwm_gpio_to_slice_num(0);
20
21 // 将周期设置为4个计数单位（包括0至3）
22 pwm_set_wrap(slice_num, 3);
23 // 设置通道A先保持高电平一个周期然后下降
24 pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
25 // 在下降前将B输出保持高电平三个周期
26 pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
27 // 启动PWM
28 pwm_set_enabled(slice_num, true);

```

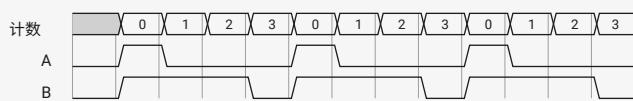
图105展示了PWM硬件在此配置方式下的运行情况。

图105。该切片计数从0计数至3，3为配置的TOP值。因此输出波形的周期为4。

输出A在4个周期中高电平持续1个周期，故平均输出电压为IO电源电压的1/4。

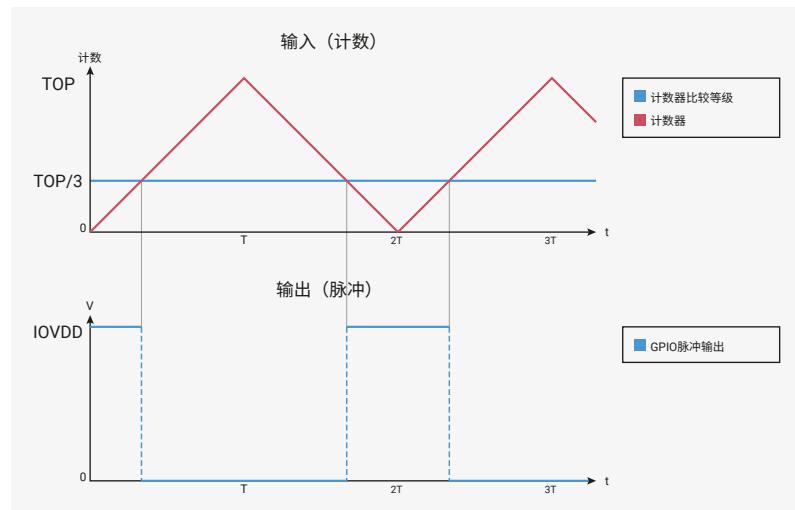
输出B在每4个周期中高电平持续3个周期。注意A和B的上升沿始终保持对齐。

图106。在相位校正模式下，计数器在达到TOP后开始从TOP向下计数至0。



PWM切片的默认行为是向上计数，直至达到 TOP寄存器的设定值，然后立即回绕至0。PWM片段还提供相位校正模式，通过将CSR_PH_CORRECT设置为1启用，该模式下计数器在达到TOP后开始向下计数，直至再次达到0。

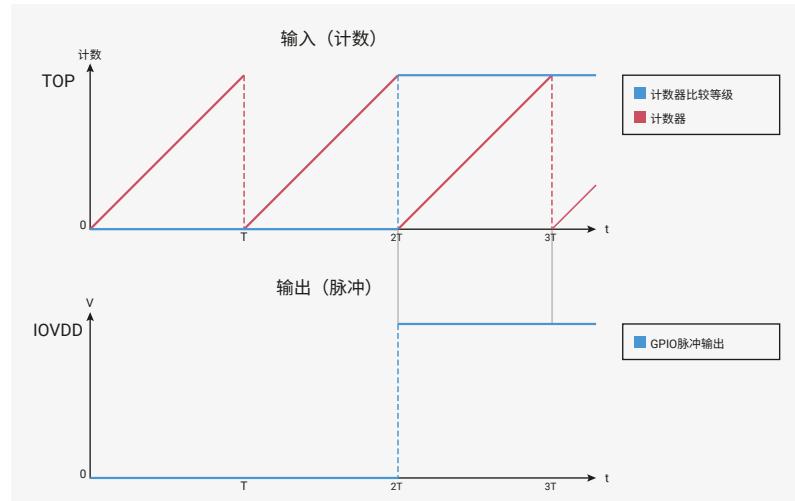
之所以称为相位校正模式，是因为脉冲始终以相同点为中心，无论占空比如何变化。换言之，其相位不依赖于占空比。启用相位校正模式时，输出频率减半。



4.5.2.2 0%和100%占空比

RP2040 PWM能够产生无跳变的0%和100%占空比输出。

图107。当CC = 0时，实现无毛刺的0%占空比输出；当CC = TOP + 1时，实现无毛刺的100%占空比输出。
1



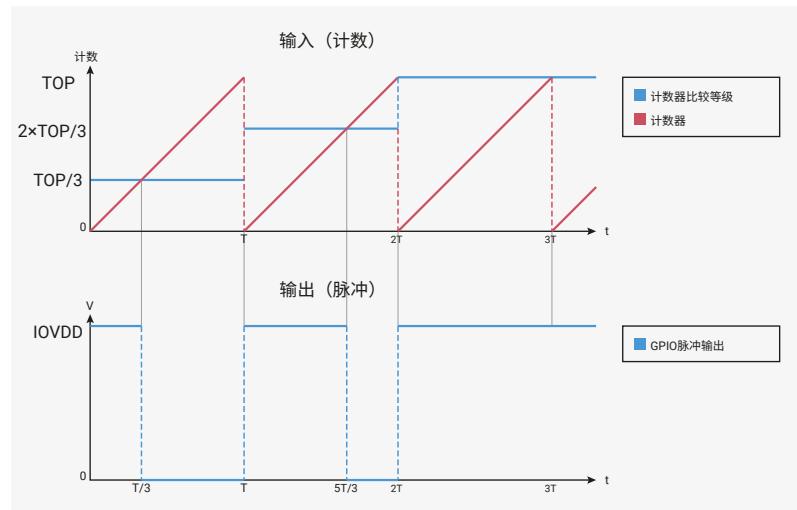
当CC值为0时，将产生0%输出，即输出信号始终为低电平；当CC值为TOP+1（即在非相位校正模式下等于周期）时，将产生100%输出。例如，若TOP设定为254，则计数器周期为255个时钟周期，且CC值在0至255之间（含）将对应0%至100%（含）的占空比。

实现0%和100%的无毛刺输出至关重要，例如避免在MOSFET以其最小和最大电流控制时产生开关损耗。

4.5.2.3. 双缓冲

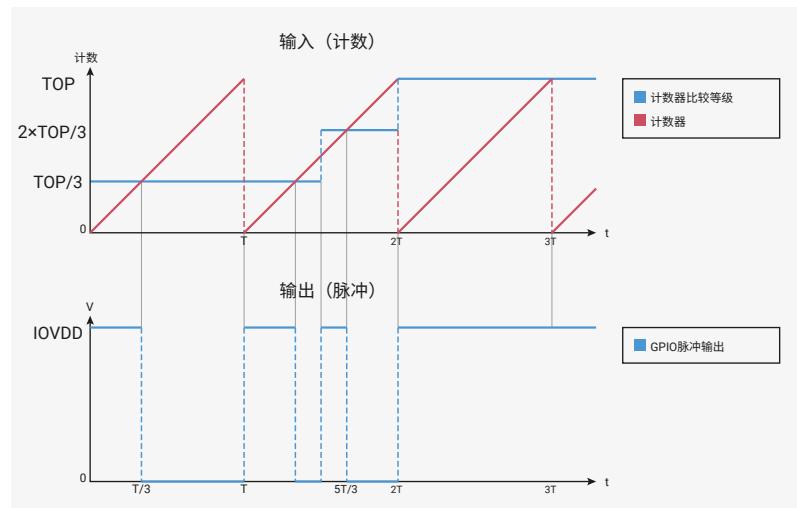
图108显示了输入值的变化如何导致输出占空比的变化。此方法可用于近似某些模拟波形，例如正弦波。

图108。输入值随每个计数周期变化：依次为 $TOP / 3$ ， $2 \times TOP / 3$ ，最后为 $TOP + 1$ ，对应100%的占空比。每次输入值的增加均会引起输出占空比的相应增加。



在图108中，输入值仅在计数器计数回零瞬间发生变化。图109展示了若允许输入值在其他任意时刻改变，输出将产生不期望的毛刺。

图109。输入值在计数器计数中途时发生变化。这将导致输出出现额外的切换。

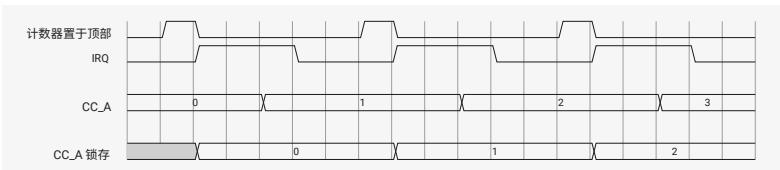


如果同时修改 TOP 寄存器，行为将更加复杂难以理解。软件难以以正确时机对 CC 或 TOP 寄存器进行写操作。为解决此问题，每个切片设有两份 CC 和 TOP 寄存器副本：一份由软件修改，另一份为内部副本，在计数器计数回零瞬间从前者更新。软件可随意更改其寄存器副本，但更改仅在下一次计数器回零时才被 PWM 输出捕获。

图110展示了软件中断处理程序在每次计数器回零时更改 CC_A 数值的事件序列。

图110。每当计数器回绕时，中断请求信号即被置位。处理器进入中断处理程序，向其CC寄存器的副本写入数据，并清除中断。

当计数器再次回绕时，CC寄存器的锁存版本会即时更新为软件最近写入的值，该值将控制下一周期的占空比。中断请求信号再次被置位，以便软件向其CC寄存器副本写入新的值。



对CC或TOP的写入数值及写入时间不受任何限制。在正常PWM模式下（CSR_PH_CORRECT为0），锁存副本在计数器回绕至0时更新，该过程每TOP+1个周期发生一次。在相位校正模式下（CSR_PH_CORRECT为1），锁存副本在0到0的计数转换点更新，即计数器停止向下计数并开始向上计数的时刻。

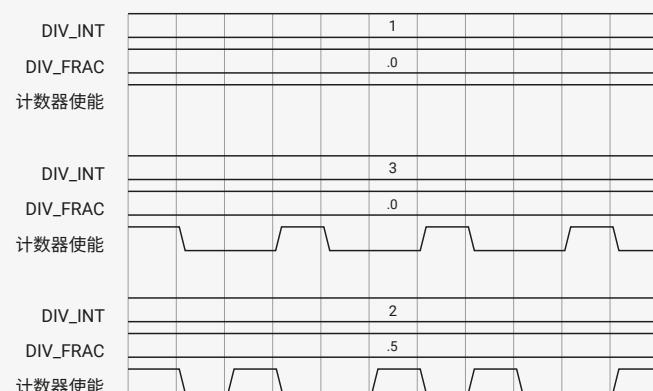
4.5.2.4. 时钟分频器

每个片段具有一个分数时钟分频器，由DIV寄存器配置。这是一个8位整数位和4位小数位的时钟分频器，允许计数速率最多降低256倍。时钟分频器可实现更低的输出频率——从125MHz系统时钟约降至7.5Hz。低于此频率将需要系统定时器中断（第4.6节）。

其工作原理是生成一个使能信号，该信号门控计数器的运作。

图111。时钟分频器生成使能信号。计数器仅在该信号为高电平时计数。分频系数为1时，使能在每个周期断言，计数器在每个系统时钟周期递增1。较高的分频系数会降低计数使能的断言频率。

分数分频通过延长部分使能脉冲间隔，实现平均分母数计数速率。

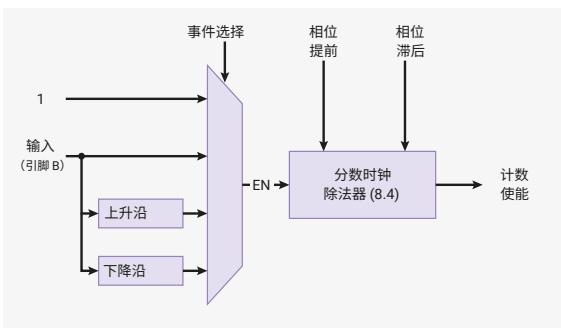


分数除法器属于一阶delta-sigma型。

时钟除法器还允许在使用电平敏感或边沿敏感模式进行占空比或频率测量时，有效计数范围得以扩展。

4.5.2.5. 电平敏感与边沿敏感触发

图 112. PWM 分片事件选择。当其使能输入为高电平时，计数器递增，该使能信号由两个连续阶段生成。首先，四种事件类型之一（始终开启、引脚 B 高电平、引脚 B 上升沿、引脚 B 下降沿）可为分数时钟除法器生成使能脉冲。除法器可降低使能脉冲的频率，再将脉冲传递给计数器。



默认情况下，每个分片的计数器均为自由运行状态，只要分片被使能，计数器将连续计数。还有另外三种可选项：

- 当B脚检测到高电平时连续计数
- 每当检测到B脚的上升沿时计数一次
- 每当检测到B脚的下降沿时计数一次

这些模式由各切片CSR寄存器中的DIVMODE字段选择。在自由运行模式下，A脚和B脚均为输出。在其他任何模式下，B脚变为输入，并控制计数器的运行。非自由运行模式下，CC_B字段被忽略。

通过允许切片在电平敏感或边缘敏感模式下运行特定时间，可测量输入信号的占空比或频率。由于所用边缘检测电路的特性，频率测量时，信号的高电平周期和低电平周期均必须严格大于系统时钟周期。

时钟分频器在电平敏感和边缘敏感模式下依然有效。在最大分频（向DIV_INT写入0）时，计数器在电平敏感模式下每256个高电平输入周期仅递增一次，或在边缘敏感模式下每256个边沿递增一次。该设置允许执行更长时间的测量，尽管分辨率仍为16位。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pwm/measure_duty_cycle/measure_duty_cycle.c 第19至37行

```

19 float measure_duty_cycle(uint gpio) {
20     // 仅PWM通道B的引脚可用于输入。
21     assert(pwm_gpio_to_channel(gpio) == PWM_CHAN_B);
22     uint slice_num = pwm_gpio_to_slice_num(gpio);
23
24     // 每当PWM通道B输入保持高电平100个周期时计数一次
25     pwm_config cfg = pwm_get_default_config();
26     pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_HIGH);
27     pwm_config_set_clkdiv(&cfg, 100);
28     pwm_init(slice_num, &cfg, false);
29     gpio_set_function(gpio, GPIO_FUNC_PWM);
30
31     pwm_set_enabled(slice_num, true);
32     sleep_ms(10);
33     pwm_set_enabled(slice_num, false);
34     float counting_rate = clock_get_hz(clk_sys) / 100;
35     float max_possible_count = counting_rate * 0.01;
36     return pwm_get_counter(slice_num) / max_possible_count;
37 }
```

4.5.2.6. 配置 PWM 周期

在自由运行模式下，PWM 片段输出的周期（以系统时钟周期计）由以下三个参数控制：

- **TOP** 寄存器
- 相位校正模式是否启用 (**CSR_PH_CORRECT**)
- **DIV** 寄存器

计数器从 0 计数到 **TOP**，然后根据 **CSR_PH_CORRECT** 的设置，要回绕，要开始反向计数。计数速率由时钟分频器降低，最大速度为每周期一次计数，

最小速度为每 $\frac{15}{255}$ 周期一次计数。时钟周期内的周期可计算为：

$$\text{period} = (\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)$$

然后可根据系统时钟频率确定输出频率：

$$f_{\text{PWM}} = \frac{f_{\text{sys}}}{\text{period}} = \frac{f_{\text{sys}}}{(\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)}$$

4.5.2.7. 中断请求 (IRQ) 与DMA数据请求 (DREQ)

PWM模块有一个单一的IRQ输出。中断状态寄存器 **INTR**、**INTS**和**INTE**允许软件控制哪些切片会触发该IRQ输出，检查引起IRQ的切片，并清除与确认中断。

每当切片计数器回绕（或启用**CSR_PH_CORRECT**时计数器返回至0）时，该切片将生成一个中断请求。这将在原始中断状态寄存器 **INTR**中设置对应该切片的标志。若该切片的中断在 **INTE**中被启用，该标志将触发PWM模块的IRQ，并且该标志亦会出现在屏蔽中断状态寄存器 **INTS**中。

通过向 **INTR**写入掩码即可清除标志。此内容已在“LED fade” SDK示例中演示。

该方案允许多个切片同时产生中断，系统中断处理程序能够确定最近期的中断由哪些切片引起，并进行相应处理。通常这意味着需要重新加载这些切片的 **TOP**或**CC**寄存器，但PWM模块也可作为非PWM相关用途的定期中断请求源。

在 **INTR**中设置中断标志的脉冲，同时也可作为RP2040系统DMA的单周期数据请求信号。DMA每检测到DREQ有效一个周期，即会尽可能及时地向其预设位置执行一次数据传输。结合 **CC**与 **TOP**的双缓冲特性，DMA能够以每个计数周期一次传输的速率，高效地向PWM切片进行数据流传输。或者，PWM切片也可用作DMA向其他内存映射硬件传输的节拍定时器。

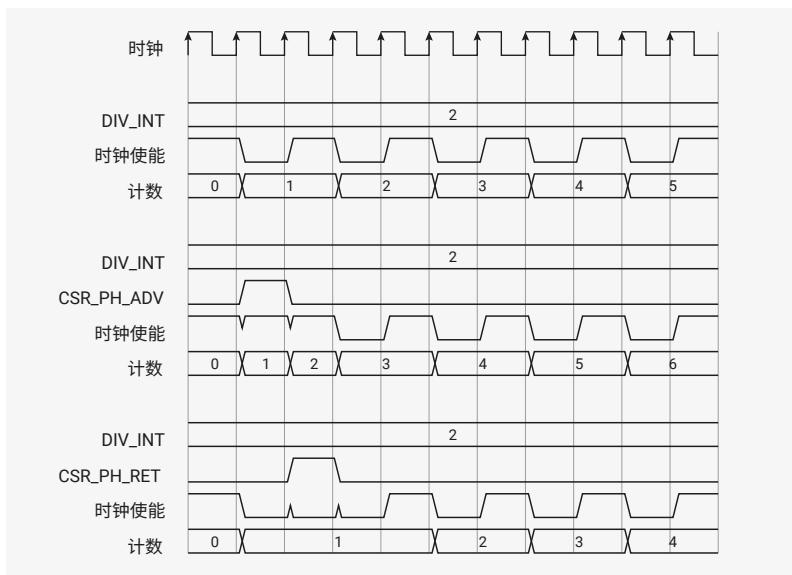
4.5.2.8. 即时相位调整

对于某些应用，需控制不同切片上两个PWM输出的相位关系。

全局使能寄存器 **EN**包含每个切片的 **CSR_EN**标志别名，允许多个切片同时启动和停止。若两个具有相同输出频率的切片同时启动，将实现完美锁步运行，并保持由初始计数器值决定的固定相位关系。

CSR_PH_ADV和**CSR_PH_RET**字段能在切片运行时使其输出相位提前或延迟一个计数。其方法是在时钟使能（即时钟分频器输出）中插入或删除脉冲，如图113所示。

图113。时钟使能信号由时钟分频器输出，用以控制计数速率。相位提前通过在时钟使能为低电平时的周期内将其强制置为高电平，使计数器跳跃前进一个计数。相位延迟使时钟使能在应为高电平时变为低电平，从而使计数器回退一个计数。



计数器不能在一个周期内计数超过一次，因此 **PH_ADV** 要求 **DIV_INT > 1** 或 **DIV_FRAC > 0**。同理，若在时钟使能持续为低电平时断言 **PH_RET**，计数器不会开始向后计数。

要使相位前进或延迟一个计数，软件需向 **PH_ADV** 或 **PH_RET** 写入1。一旦插入或删除一个使能脉冲，**PH_ADV** 或 **PH_RET** 寄存器位将恢复为0，软件可轮询 **CSR** 直至此状态。**PH_ADV** 总是在下一个可用空隙插入一个脉冲，而 **PH_RET** 总是删除下一个可用脉冲。

4.5.3. 寄存器列表

PWM寄存器起始地址为 **0x40050000** (SDK中定义为PWM_BASE)。

表516。
PWM寄存器列表

偏移量	名称	说明
0x00	CH0_CSR	控制与状态寄存器
0x04	CH0_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x08	CH0_CTR	直接访问 PWM 计数器
0x0c	CH0_CC	计数器比较值
0x10	CH0_TOP	计数器回绕值
0x14	CH1_CSR	控制与状态寄存器
0x18	CH1_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x1c	CH1_CTR	直接访问 PWM 计数器
0x20	CH1_CC	计数器比较值
0x24	CH1_TOP	计数器回绕值
0x28	CH2_CSR	控制与状态寄存器
0x2c	CH2_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。

偏移量	名称	说明
0x30	CH2_CTR	直接访问 PWM 计数器
0x34	CH2_CC	计数器比较值
0x38	CH2_TOP	计数器回绕值
0x3c	CH3_CSR	控制与状态寄存器
0x40	CH3_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x44	CH3_CTR	直接访问 PWM 计数器
0x48	CH3_CC	计数器比较值
0x4c	CH3_TOP	计数器回绕值
0x50	CH4_CSR	控制与状态寄存器
0x54	CH4_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x58	CH4_CTR	直接访问 PWM 计数器
0x5c	CH4_CC	计数器比较值
0x60	CH4_TOP	计数器回绕值
0x64	CH5_CSR	控制与状态寄存器
0x68	CH5_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x6c	CH5_CTR	直接访问 PWM 计数器
0x70	CH5_CC	计数器比较值
0x74	CH5_TOP	计数器回绕值
0x78	CH6_CSR	控制与状态寄存器
0x7c	CH6_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x80	CH6_CTR	直接访问 PWM 计数器
0x84	CH6_CC	计数器比较值
0x88	CH6_TOP	计数器回绕值
0x8c	CH7_CSR	控制与状态寄存器
0x90	CH7_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x94	CH7_CTR	直接访问 PWM 计数器
0x98	CH7_CC	计数器比较值
0x9c	CH7_TOP	计数器回绕值

偏移量	名称	说明
0xa0	EN	该寄存器为所有通道的 CSR_EN 位设置别名。 写入此寄存器可同时启用或禁用多个通道，实现完美同步运行。 每个通道仅有一个物理 EN 寄存器位， 可通过此处或 CHx_CSR 访问。
0xa4	INTR	原始中断
0xa8	INTE	中断使能
0xac	INTF	中断强制
0xb0	INTS	掩码及强制后的中断状态

PWM: CH0_CSR、CH1_CSR、...、CH6_CSR、CH7_CSR 寄存器

偏移量: 0x00, 0x14, ..., 0x78, 0x8c

描述

控制与状态寄存器

表 517。CH0_CSR、CH1_CSR、...、CH6_CSR、CH7_CSR
R 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	PH_ADV : 在计数器运行时，将计数器相位提前 1 个计数。 自清零。写入1，并轮询直至变为低电平。计数器必须以低于满速 (div_int + div_frac / 16 > 1) 运行。	SC	0x0
6	PH_RET : 在计数器运行时，将其相位延迟1个计数。 自清零。写入1，并轮询直至变为低电平。计数器必须处于运行状态。	SC	0x0
5:4	DIVMODE	读写	0x0
	枚举值：		
	0x0 → DIV：以分数分频器确定的速率自由运行计数。		
	0x1 → LEVEL：分数分频器操作受PWM B引脚门控。		
	0x2 → RISE：计数器随PWM B引脚的每个上升沿递增。		
	0x3 → FALL：计数器随PWM B引脚的每个下降沿递增。		
3	B_INV : 反转输出B。	读写	0x0
2	A_INV : 反转输出A。	读写	0x0
1	PH_CORRECT : 1: 启用相位校正调制；0: 后沿调制。	读写	0x0
0	EN : 启用PWM通道。	读写	0x0

PWM: CH0_DIV、CH1_DIV、.....、CH6_DIV、CH7_DIV 寄存器

偏移量: 0x04、0x18、.....、0x7c、0x90

描述

INT 和 FRAC 组成一个定点小数部分。
计数速率为系统时钟频率除以该数值。
分数除法采用简单的一阶Σ-Δ调制。

表 518。CH0_DIV、CH1_DIV、.....CH6_DIV
、CH7_DIV 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:4	中断	读写	0x01
3:0	FRAC	读写	0x0

PWM：CH0_CTR、CH1_CTR、.....CH6_CTR、CH7_CTR 寄存器

偏移量：0x08、0x1c、.....0x80、0x94

表 519。CH0_CTR、CH1_CTR、.....CH6_CTR、CH7_CTR 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	直接访问 PWM 计数器	读写	0x0000

PWM：CH0_CC、CH1_CC、.....CH6_CC、CH7_CC 寄存器

偏移量：0x0c、0x20、.....0x84、0x98

描述

计数器比较值

表 520。CH0_CC、CH1_CC、.....CH6_CC
、CH7_CC 寄存器

位	描述	类型	复位值
31:16	B	读写	0x0000
15:0	A	读写	0x0000

PWM：CH0_TOP、CH1_TOP、...、CH6_TOP、CH7_TOP 寄存器

偏移量：0x10、0x24、...、0x88、0x9c

表 521。CH0_TOP、CH1_TOP、...、CH6_TOP、CH7_TOP
P 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	计数器回绕值	读写	0xffff

PWM：EN 寄存器

偏移：0xa0

描述

该寄存器为所有通道的 CSR_EN 位设置别名。

写入此寄存器可同时启用或禁用多个通道，实现完美同步运行。

每个通道仅有一个物理 EN 寄存器位，
可通过此处或 CHx_CSR 访问。

表 522。EN 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0

位	描述	类型	复位值
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTR寄存器

偏移: 0xa4

描述

原始中断

表 523. INTR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	WC	0x0
6	CH6	WC	0x0
5	CH5	WC	0x0
4	CH4	WC	0x0
3	CH3	WC	0x0
2	CH2	WC	0x0
1	CH1	WC	0x0
0	CH0	WC	0x0

PWM: INTF 寄存器

偏移: 0xac

描述

中断使能

表 524. INTF
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTE 寄存器

偏移: 0xa8

描述

中断强制

表 525. INTF
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTS 寄存器

偏移: 0xb0

描述

掩码及强制后的中断状态

表 526. INTS
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	只读	0x0
6	CH6	只读	0x0
5	CH5	只读	0x0
4	CH4	只读	0x0
3	CH3	只读	0x0
2	CH2	只读	0x0
1	CH1	只读	0x0
0	CH0	只读	0x0

4.6. 定时器**4.6.1. 概述**

RP2040上的系统定时器外设为系统提供全局微秒时间基准，并基于该时间基准产生中断。其支持以下功能：

- 一个64位单计数器，每微秒递增一次
- 该计数器可通过一对锁存寄存器读取，实现基于32位总线的无竞争读取。
- 四个报警：匹配计数器低32位时触发中断请求（IRQ）。

该定时器采用在看门狗中生成的一微秒参考时钟（详见第4.7.2节）作为基准，并由此派生。

参考时钟（图28），通常直接连接至晶体振荡器（第2.16节）。

64位计数器实际上不会溢出（以1MHz计时，持续数千年），因此系统定时器在实际应用中是完全单调递增的。

4.6.1.1. RP2040上的其他定时器资源

系统定时器旨在为软件提供全局时基。RP2040具有多种其他可编程计数器资源，能够提供定期中断或触发DMA传输。

- PWM（第4.5节）包含8个16位可编程计数器，最高运行速度可达系统时钟频率，能够产生中断，且可通过DMA持续重新编程，或触发对其他外设的DMA传输。
- 8个PIO状态机（第3章）能够以系统速度计数32位值并产生中断。
- DMA（第2.5节）具有四个内部节拍计时器，可定时触发传输。
- 每个 Cortex-M0+ 核心（见第 2.4 节）均配备标准的 24 位 SysTick 定时器，可计数微秒滴答（见第 4.7.2 节）或系统时钟。

4.6.2. 计数器

该定时器拥有 64 位计数器，但 RP2040 仅具备 32 位数据总线。这意味着 **TIME** 值需通过一对寄存器访问，具体如下：

- **TIMEHW** 和 **TIMELW** 用于写入时间
- **TIMEHR** 和 **TIMELR** 用于读取时间

访问该对寄存器时，先访问低位寄存器 **L**，随后访问高位寄存器 **H**。读取时，读取 **L** 寄存器会锁存 **H** 寄存器的数值，确保时间读取的准确性。或者，可使用 **TIMERAWH** 和 **TIMERAWL** 来读取未经过锁存的原始时间。

⚠ 注意

尽管技术上可通过向 **TIMEHW** 和 **TIMELW** 寄存器写入来强制设置新的时间值，但不建议程序员这样操作。这是因为计时器数值被SDK预期为单调递增，SDK利用该值实现超时、计时等功能。

4.6.3. 报警

该计时器拥有4个报警，每个报警对应一个独立中断输出。报警基于64位计数器的低32位进行匹配，意味着报警可在 2^{32} 微秒内触发。相当于：

- $2^{32} \div 10^6$ ：约4295秒
- $4295 \div 60$ ：约72分钟

ℹ 注意

此计时器适用于短时休眠。如需更长时间报警，请参见第4.8节。

启用报警步骤如下：

- 通过向**INTE** 中对应报警位写入值来启用计时器中断：如(**1 << 0**)表示启用 **ALARM0**
- 在处理器端启用相应的计时器中断（详见第2.3.2节）
- 将希望触发中断的时间写入**ALARM0**（即当前**TIMERAWL**值加上所需的报警时间，单位为微秒）。向 **ALARM**寄存器写入时间时，会作为副作用设置**ARMED**位。

闹钟触发后，ARMED位将被设置为 0。要清除锁存中断，请向INTR中的相应位写入 1。

4.6.4. 程程序员模型

① 注意

看门狗计时器（参见第4.7.2节）必须运行，定时器才能开始计数。SDK在平台初始化代码中启动该计时器。

4.6.4.1 读取时间

① 注意

此处的时间指定时器启动后经过的微秒数，非时钟时间。有关时钟时间，请参见第4.8节。

读取64位时间的最简方式是先读取TIMELR，再读取TIMEHR。但鉴于RP2040拥有双核心，若第二核心执行的代码可访问定时器，或在IRQ处理程序与线程模式中并发读取定时器，则此方法不安全。这是因为读取 TIMELR 会锁存 TIMEHR 中的值（即停止其更新），直到 TIMEHR 被读取。若一个核心读取 TIMELR，随后另一个核心也读取 TIMELR，则 TIMEHR 中的值未必准确。以下示例展示了获取 64 位时间的最简形式。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c 第 15 至 23 行

```
15 // 从计时器获取 64 位时间的最简形式。
16 // 由于锁存机制，多个核心调用时不安全
17 // 因此 SDK 中未采用此实现方式
18 static uint64_t get_time(void) {
19     // 读取低位时锁存高位值
20     uint32_t lo = timer_hw->timelr;
21     uint32_t hi = timer_hw->timehr;
22     return ((uint64_t) hi << 32u) | lo;
23 }
```

SDK 提供了一个 `time_us_64` 函数，采用更完善的方法获取64位时间，该方法利用了 TIMERAWH 和 TIMERAWL 寄存器。由于 `RAW` 寄存器不具备锁存功能，故 `ime_us_64` 函数可安全地被多个核心同时调用。

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c 第57至73行

```
57 uint64_t timer_time_us_64(timer_hw_t *timer) {
58     // 需确保计时器高32位
59     // 不发生变化，故先读取高位
60     uint32_t hi = timer->timerawh;
61     uint32_t lo;
62     do {
63         // 读取低32位
64         lo = timer->timerawl;
65         // 现再次读取高32位，并
66         // 检查计数器是否未递增。如已递增，则循环
67         // 并再次读取低32位，以获取准确值
68         uint32_t next_hi = timer->timerawh;
69         if (hi == next_hi) break;
70         hi = next_hi;
```

```

71     } while (true);
72     return ((uint64_t) hi << 32u) | lo;
73 }

```

4.6.4.2. 设置闹钟

独立计时器示例 timer_lowlevel 展示了如何在硬件级别设置闹钟，而无需使用 SDK 提供的计时器抽象层。有关使用这些抽象层的详细说明，请参见第 4.6.4.4 节。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c, 第 27 行至第 74 行

```

27 // 使用闹钟 0
28 #define ALARM_NUM 0
29 #define ALARM_IRQ timer_hw_alarm_get_irq_num(timer_hw, ALARM_NUM)
30
31 // 警报中断处理程序
32 static volatile bool alarm_fired;
33
34 static void alarm_irq(void) {
35     // 清除警报中断
36     hw_clear_bits(&timer_hw->intr, 1u << ALARM_NUM);
37
38     // 假设警报0已触发
39     printf("Alarm IRQ fired\n");
40     alarm_fired = true;
41 }
42
43 static void alarm_in_us(uint32_t delay_us) {
44     // 启用我们的警报中断（定时器输出4个警报中断）
45     hw_set_bits(&timer_hw->inte, 1u << ALARM_NUM);
46     // 设置警报中断处理程序
47     irq_set_exclusive_handler(ALARM_IRQ, alarm_irq);
48     // 启用警报中断
49     irq_set_enabled(ALARM_IRQ, true);
50     // 在模块和处理器处启用中断
51
52     // 警报仅为32位，如果尝试延迟超过该值，
53     // 则需谨慎并跟踪高位
54     // 位
55     uint64_t target = timer_hw->timerawl + delay_us;
56
57     // 将目标时间的低32位写入警报寄存器以启动警报
58     // will arm it
59     timer_hw->alarm[ALARM_NUM] = (uint32_t) target;
60 }
61
62 int main() {
63     stdio_init_all();
64     printf("Timer lowlevel!\n");
65
66     // 每隔2秒设置一次警报
67     while (1) {
68         alarm_fired = false;
69         alarm_in_us(1000000 * 2);
70         // 等待警报触发
71         while (!alarm_fired);
72     }
73 }

```

4.6.4.3. 忙等待

如果您不想使用闹钟等待一段时间，可以改用 while 循环。SDK 提供了多种 `busy_wait_` 函数用于此目的：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c 第77至122行

```

77 void timer_busy_wait_us_32(timer_hw_t *timer, uint32_t delay_us) {
78     if (0 <= (int32_t)delay_us) {
79         // 仅允许使用31位，否则接近2^32的值在下面的循环中可能引发竞态条件
80         // values very close to 2^32
81         uint32_t start = timer->timerawl;
82         while (timer->timerawl - start < delay_us) {
83             tight_loop_contents();
84         }
85     } else {
86         busy_wait_us(delay_us);
87     }
88 }
89
90 void timer_busy_wait_us(timer_hw_t *timer, uint64_t delay_us) {
91     uint64_t base = timer_time_us_64(timer);
92     uint64_t target = base + delay_us;
93     if (target < base) {
94         target = (uint64_t)-1;
95     }
96     absolute_time_t t;
97     update_us_since_boot(&t, target);
98     timer_busy_wait_until(timer, t);
99 }
100
101 void timer_busy_wait_ms(timer_hw_t *timer, uint32_t delay_ms)
102 {
103     if (delay_ms <= 0x7fffffff / 1000) {
104         timer_busy_wait_us_32(timer, delay_ms * 1000);
105     } else {
106         timer_busy_wait_us(timer, delay_ms * 1000ull);
107     }
108 }
109
110 void timer_busy_wait_until(timer_hw_t *timer, absolute_time_t t) {
111     uint64_t target = to_us_since_boot(t);
112     uint32_t hi_target = (uint32_t)(target >> 32u);
113     uint32_t hi = timer->timerawh;
114     while (hi < hi_target) {
115         hi = timer->timerawh;
116         tight_loop_contents();
117     }
118     while (hi == hi_target && timer->timerawl < (uint32_t) target) {
119         hi = timer->timerawh;
120         tight_loop_contents();
121     }
122 }
```

4.6.4.4. 使用 SDK 的完整示例

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/timer/hello_timer/hello_timer.c 第11至57行

```

11 volatile bool timer_fired = false;
12
13 int64_t alarm_callback(alarm_id_t id, __unused void *user_data) {
14     printf("定时器%d触发! \n", (int) id);
15     timer_fired = true;
16     // 可返回一个微秒数值用于将来触发
17     return 0;
18 }
19
20 bool repeating_timer_callback(__unused struct repeating_timer *t) {
21     printf("重复触发时间: %lld\n", time_us_64());
22     return true;
23 }
24
25 int main() {
26     stdio_init_all();
27     printf("你好, 定时器! \n");
28
29     // 两秒后调用 alarm_callback
30     add_alarm_in_ms(2000, alarm_callback, NULL, false);
31
32     // 等待闹钟回调以设置 timer_fired
33     while (!timer_fired) {
34         tight_loop_contents();
35     }
36
37     // 创建一个重复定时器, 调用 repeating_timer_callback。
38 // 如果延迟 > 0, 则表示前一次回调结束到下一次回调开始之间的延迟。
39 // 如果延迟为负值 (详见下文), 则下一次回调将精确地在上一次回调开始 500ms 后触发
40
41     // start of the call to the last callback
42     struct repeating_timer timer;
43     add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
44     sleep_ms(3000);
45     bool cancelled = cancel_repeating_timer(&timer);
46     printf("cancelled... %d\n", cancelled);
47     sleep_ms(2000);
48
49     // 负延迟意味着我们会调用 repeating_timer_callback, 并随后再次调用
50     // 无论回调执行时间长短, 500 毫秒后触发
51     add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
52     sleep_ms(3000);
53     cancelled = cancel_repeating_timer(&timer);
54     sleep_ms(2000);
55     printf("Done\n");
56     return 0;
57 }
```

4.6.5. 寄存器列表

定时器寄存器起始地址为 `0x40054000` (在 SDK 中定义为 `TIMER_BASE`)。

表 527.
定时器寄存器列表

偏移量	名称	说明
0x00	TIMEHW	写入 time 的 63:32 位 必须先写入 timelw, 然后写入 timehw

偏移量	名称	说明
0x04	TIMELW	写入 time 的 31:0 位 写入操作在写入 timehw 前不会复制到 time
0x08	TIMEHR	读取 time 的 63:32 位 必须先读取 timelr，然后读取 timehr
0x0c	TIMELR	读取 time 的 31:0 位
0x10	ALARM0	使能报警器0，并配置其触发时间。 使能后，当 TIMER_ALARM0 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x14	报警器1	使能报警器1，并配置其触发时间。 使能后，当 TIMER_ALARM1 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x18	报警器2	使能报警器2，并配置其触发时间。 使能后，当 TIMER_ALARM2 == TIMELR 时报警器触发。报警 器触发后将自动解除使能，也可通过 ARMED 状 态寄存器提前解除使能。
0x1c	报警器3	使能报警器3，并配置其触发时间。 使能后，当 TIMER_ALARM3 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x20	ARMED	指示各报警器的使能/解除使能状态。 向对应的 ALARMx 寄存器写入数据即使能该报警器。 报警触发后会自动解除使能，写入 1 可立即解除，无需等待触 发。
0x24	TIMERAWH	对时间的第63至32位进行原始读取（无副作用）
0x28	TIMERAWL	对时间的第31至0位进行原始读取（无副作用）
0x2c	DBGPAUSE	将对应位设为高电平以启用调试端口激活时的暂停
0x30	PAUSE	置高以暂停计时器
0x34	INTR	原始中断
0x38	INTE	中断使能
0x3c	INTF	中断强制
0x40	INTS	掩码及强制后的中断状态

TIMER: TIMEHW 寄存器

偏移: 0x00

表 528. TIMEHW
寄存器

位	描述	类型	复位值
31:0	写入 time 的 63:32 位 必须先写入 timelw，然后写入 timehw	WF	0x00000000

TIMER：TIMELW 寄存器

偏移: 0x04

表 529. TIMELW
寄存器

位	描述	类型	复位值
31:0	写入 time 的 31:0 位 写入操作在写入 timehw 前不会复制到 time	WF	0x00000000

TIMER：TIMEHR 寄存器

偏移: 0x08

表 530. TIMEHR
寄存器

位	描述	类型	复位值
31:0	读取 time 的 63:32 位 必须先读取 timelr，然后读取 timehr	只读	0x00000000

TIMER：TIMELR 寄存器

偏移: 0x0c

表 531. TIMELR
寄存器

位	描述	类型	复位值
31:0	读取 time 的 31:0 位	只读	0x00000000

TIMER：ALARM0 寄存器

偏移: 0x10

表 532. ALARM0
寄存器

位	描述	类型	复位值
31:0	使能报警器0，并配置其触发时间。 使能后，当 TIMER_ALARM0 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM1 寄存器

偏移量: 0x14

表 533. ALARM1
寄存器

位	描述	类型	复位值
31:0	使能报警器1，并配置其触发时间。 使能后，当 TIMER_ALARM1 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM2 寄存器

偏移: 0x18

表 534. ALARM2
寄存器

位	描述	类型	复位值
31:0	使能报警器2，并配置其触发时间。 使能后，当 TIMER_ALARM2 == TIMELR 时报警器触发。报警器触发后将自动解除使能，也可通过 ARMED 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM3 寄存器

偏移量: 0x1c

表 535. ALARM3 寄存器

位	描述	类型	复位值
31:0	使能报警器3，并配置其触发时间。 使能后，当 TIMER_ALARM3 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ARMED 寄存器

偏移量: 0x20

表 536. ARMED 寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	指示各报警器的使能/解除使能状态。 向对应的 ALARMx 寄存器写入数据即使能该报警器。 报警触发后会自动解除使能，写入 1 可立即解除，无需等待触发。	WC	0x0

定时器：TIMERAWH 寄存器

偏移：0x24

表 537. TIMERAWH 寄存器

位	描述	类型	复位值
31:0	对时间的第63至32位进行原始读取（无副作用）	只读	0x00000000

定时器：TIMERAWL 寄存器

偏移：0x28

表 538. TIMERAWL 寄存器

位	描述	类型	复位值
31:0	对时间的第31至0位进行原始读取（无副作用）	只读	0x00000000

定时器：DBGPAUSE 寄存器

偏移量: 0x2c

描述

将对应位设为高电平以在相应调试端口激活时启用暂停

表 539. DBGPAUSE 寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	DBG1 : 当处理器 1 处于调试模式时暂停	读写	0x1
1	DBG0 : 当处理器0处于调试模式时暂停	读写	0x1

位	描述	类型	复位值
0	保留。	-	-

TIMER: PAUSE 寄存器

偏移量: 0x30

表540. PAUSE
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	置高以暂停计时器	读写	0x0

TIMER: INTR 寄存器

偏移量: 0x34

描述

原始中断

表541. INTR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	WC	0x0
2	ALARM_2	WC	0x0
1	ALARM_1	WC	0x0
0	ALARM_0	WC	0x0

TIMER: INTE 寄存器

偏移量: 0x38

描述

中断使能

表542. INTE
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	读写	0x0
2	ALARM_2	读写	0x0
1	ALARM_1	读写	0x0
0	ALARM_0	读写	0x0

TIMER: INTF 寄存器

偏移量: 0x3c

描述

中断强制

表543. INTF
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	读写	0x0

位	描述	类型	复位值
2	ALARM_2	读写	0x0
1	ALARM_1	读写	0x0
0	ALARM_0	读写	0x0

TIMER: INTS 寄存器

偏移: 0x40

描述

掩码及强制后的中断状态

表544. INTS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	只读	0x0
2	ALARM_2	只读	0x0
1	ALARM_1	只读	0x0
0	ALARM_0	只读	0x0

4.7. 看门狗

4.7.1. 概述

看门狗是一种倒计时定时器，计数到零时可重启芯片的部分模块。该功能可用于在软件陷入无限循环时重启处理器。程序员必须定期向看门狗写入值，以防其计数归零。

看门狗通过 `rst_n_run` 复位，该信号在数字核心电源（DVDD）通电且稳定且RUN引脚为高电平后立即解除断言。这使得看门狗复位信号能够传入上电状态机（参见第2.13节）和复位控制器（参见第2.14节），如果它们在 `WDSEL` 寄存器中被选中，则重置其从属部分。`WDSEL` 寄存器存在于上电状态机和复位控制器中。

4.7.2. 时钟节拍生成

看门狗参考时钟 `clk_tick` 由 `clk_ref` 驱动。理想情况下，`clk_ref` 将配置为使用晶体振荡器（第2.16节），以提供精确的参考时钟。参考时钟在内部被分频以生成一个定时脉冲（标称1μs），用作看门狗时钟脉冲。该时钟脉冲通过TICK寄存器进行配置。

① 注意

为避免重复逻辑，该时钟脉冲也分发给计时器（参见第4.6节），并用作计时器参考时钟。

SDK 在 `clocks_init` 中启动看门狗计时：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第16至18行

```
16 void watchdog_start_tick(uint cycles) {
17     tick_start(TICK_WATCHDOG, cycles);
```

18 }

4.7.3. 看门狗计数器

看门狗计数器由 LOAD 寄存器加载。当前数值可在 CTRL.TIME 中查看。

● 警告

由于逻辑错误，看门狗计数器每个计时周期递减两次。这意味着程序员需要编写两倍于预期的倒计时值。SDK示例已考虑该问题。详见 RP2040-E1 获取更多信息。

4.7.4. 暂存寄存器

看门狗包含八个32位存储寄存器，可用于在芯片软复位期间保存信息。切换 RUN 引脚或循环数字核心电源（DVDD）触发的 `rst_n_run` 事件将重置这些存储寄存器。

启动ROM在启动时检查看门狗擦写寄存器中的魔术数字。此功能可用于软重置芯片，以运行用户指定的代码。详见第2.8.1.1节。

4.7.5. 程序员模型

SDK提供hardware_watchdog驱动程序以控制看门狗。

4.7.5.1. 启用看门狗

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第42至74行

```

42 // watchdog_enable 和 watchdog_reboot 共用的辅助函数
43 void _watchdog_enable(uint32_t delay_ms, bool pause_on_debug) {
44     valid_params_if(HARDWARE_WATCHDOG, delay_ms <= WATCHDOG_LOAD_BITS / (1000 *
45         WATCHDOG_XFACTOR));
46     hw_clear_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
47
48     // 重置除ROSC和XOSC之外的所有部分
49     hw_set_bits(&psm_hw->wdsel, PSM_WDSEL_BITS & ~(PSM_WDSEL_ROSC_BITS |
50         PSM_WDSEL_XOSC_BITS));
51
52     uint32_t dbg_bits = WATCHDOG_CTRL_PAUSE_DBG0_BITS |
53                         WATCHDOG_CTRL_PAUSE_DBG1_BITS |
54                         WATCHDOG_CTRL_PAUSE_JTAG_BITS;
55
56     if (pause_on_debug) {
57         hw_set_bits(&watchdog_hw->ctrl, dbg_bits);
58     } else {
59         hw_clear_bits(&watchdog_hw->ctrl, dbg_bits);
60     }
61
62     if (!delay_ms) {
63         hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_TRIGGER_BITS);
64     } else {
65         load_value = delay_ms * 1000;
66         load_value *= 2;
67     }
68 }
```

```

65     if (load_value > WATCHDOG_LOAD_BITS)
66         load_value = WATCHDOG_LOAD_BITS;
67
68     watchdog_update();
69
70     hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
71 }
72 }
```

4.7.5.2. 更新看门狗计数器

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第24至28行

```

24 static uint32_t load_value;
25
26 void watchdog_update(void) {
27     watchdog_hw->load = load_value;
28 }
```

4.7.5.3. 使用方法

Pico 示例仓库提供了 hello_watchdog 示例，利用 hardware_watchdog 演示了看门狗的使用。

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/watchdog/hello_watchdog/hello_watchdog.c 第11至33行

```

11 int main() {
12     stdio_init_all();
13
14     if (watchdog_enable_caused_reboot()) {
15         printf("由看门狗重启! \n");
16         return 0;
17     } else {
18         printf("Clean boot\n");
19     }
20
21 // 启用看门狗，要求每100毫秒更新一次，否则芯片将重启
22
23     // 第二个参数为调试时暂停，表示单步调试时看门狗会暂停
24     // 代码
25     watchdog_enable(100, 1);
26
27     for (uint i = 0; i < 5; i++) {
28         printf("正在更新看门狗 %d\n", i);
29         watchdog_update();
30     }
31
32     // 在无限循环中等待且不更新看门狗，导致重启
33     printf("等待看门狗重启中\n");
34     while(1);
35 }
```

4.7.6. 寄存器列表

看门狗寄存器起始地址为 **0x40058000** (在SDK中定义为WATCHDOG_BASE)。

表545。 WATCH
DOG 寄存器列表

偏移量	名称	说明
0x00	CTRL	看门狗控制
0x04	LOAD	加载看门狗计时器。
0x08	REASON	记录上次复位的原因。
0x0c	SCRATCH0	暂存寄存器
0x10	SCRATCH1	暂存寄存器
0x14	SCRATCH2	暂存寄存器
0x18	SCRATCH3	暂存寄存器
0x1c	SCRATCH4	暂存寄存器
0x20	SCRATCH5	暂存寄存器
0x24	SCRATCH6	暂存寄存器
0x28	SCRATCH7	暂存寄存器
0x2c	TICK	控制滴答计时器

看门狗： CTRL寄存器

偏移: 0x00

说明

看门狗控制
rst_wdsel寄存器决定在看门狗触发时重置哪些子系统。
看门狗可由软件触发。

表 546. CTRL
寄存器

位	描述	类型	复位值
31	触发: 触发看门狗复位	SC	0x0
30	启用: 未启用时, 看门狗定时器暂停	读写	0x0
29:27	保留。	-	-
26	PAUSE_DBG1 : 当处理器1处于调试模式时暂停看门狗定时器	读写	0x1
25	PAUSE_DBG0 : 当处理器0处于调试模式时暂停看门狗定时器	读写	0x1
24	PAUSE_JTAG : 当JTAG访问总线时暂停看门狗定时器 结构	读写	0x1
23:0	TIME : 指示在触发看门狗复位前的滴答数除以2 (参见勘误 RP2040-E1)	只读	0x000000

WATCHDOG： LOAD 寄存器

偏移: 0x04

表547. LOAD
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	加载看门狗定时器。最大设置值为 0xffffffff，表示触发看门狗复位前的滴答数为 $0xffffffff / 2$ （参见勘误 RP2040-E1）。	WF	0x000000

WATCHDOG：REASON 寄存器

偏移量: 0x08

说明

记录上次复位的原因。硬件复位时，两位均为零。

表548. REASON
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	强制	只读	0x0
0	TIMER	只读	0x0

WATCHDOG：SCRATCH0、SCRATCH1、...、SCRATCH6、SCRATCH7 寄存器

偏移量: 0x0c, 0x10, ..., 0x24, 0x28

表 549。 SCRATCH0、S
CRATCH1、...
、 SCRATCH
6、 SCRATCH7 寄存器

位	描述	类型	复位值
31:0	临时寄存器。信息在芯片软复位后仍然保留。	读写	0x00000000

看门狗：TICK 寄存器

偏移量: 0x2c

描述

控制滴答计时器

表 550。 TICK
寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:11	计数：递减计时器——下一个时钟滴答生成前剩余的 clk_tick 周期数。	只读	-
10	运行中：时钟滴答生成器是否正在运行？	只读	-
9	启用：启动 / 停止时钟滴答生成	读写	0x1
8:0	周期数：下一个时钟滴答生成前的总 clk_tick 周期数。	读写	0x000

4.8. 实时时钟 (RTC)

实时时钟 (RTC) 提供人类可读格式的时间，可用于在特定时间生成中断。

4.8.1. 存储格式

时间以二进制存储，分为七个字段：

表 551。RTC
存储格式

日期/时间字段	大小	合法值
年份	12位	0..4095
月份	4位	1..12
日期	5位	1..[28,29,30,31]，具体取决于月份
星期几	3位	0..6，星期日为0
小时	5位	0..23
分钟	6位	0..59
秒	6位	0..59

RTC不会验证所编程数值是否处于有效范围内。非法数值可能导致异常行为。

4.8.1.1. 星期几

星期几按周日为0，周一为1，...，周六为6编码（即ISO8601模7）。

无内置日历功能。RTC不会计算正确的星期几；它仅会递增现有的数值。

4.8.2. 闰年

如果SETUP_0中的 **YEAR**当前值能被4整除，则判定为闰年，2月28日之后为2月29日，而非3月1日。由于这一规则并非总是适用（例如世纪年），因此可通过设置CTRL.FORCE_NOTLEAPYEAR强制关闭闰年检测。

注意

闰年检测仅在必要时执行（即2月28日23:59:59后的第二秒）。软件可在2096年3月1日00:00:00之后任意时间设置**FORCE_NOTLEAPYEAR**，前提是必须在2100年2月28日23:59:59之前完成（考虑时钟域跨越延迟）。

4.8.3. 中断

RTC可以在设定时间生成中断。IRQ_SETUP_0中设有全局位 **MATCH_ENA**以启用该功能，并针对各时间字段（年、月、日、星期、小时、分钟、秒）设有独立的使能位。该功能可用于按照指定时间实现重复中断。

闹钟中断会发送至处理器，同时也发送至ROSC和XOSC，以唤醒它们的休眠模式。有关休眠模式的详细信息，请参见第4.8.5.5节。

4.8.4. 参考时钟

RTC使用参考时钟 **clk_rtc**，频率应为1至65536Hz范围内的任意整数。

内部1Hz参考时钟由内部时钟分频器生成，该分频器将 **clk_rtc**除以一个整数值。分频值减1的结果设置在CLKDIV_M1寄存器中。

● 警告

虽然可以在RTC启用时更改CLKDIV_M1，但不建议这样操作。

`clk_rtc`可由内部或外部时钟源驱动。这些时钟源可通过分数分频器进行预分频（参见第2.15节）。

可能的时钟源示例包括：

- XOSC @ 12MHz / 256 = 46875Hz。要获得1Hz参考信号，应将CLKDIV_M1设置为46874。
- 来自GPS的外部参考信号，每秒产生一个脉冲。配置 `clk_rtc`以使用GPIO引脚20的GPIO0时钟源。在此情况下，`clk_rtc`分频器为1，内部RTC时钟分频器也为1（即CLKDIV_M1 = 0）。

i 注意

所有RTC寄存器的读写均在处理器时钟域 `clk_sys`内完成。所有数据均在不同时钟域间双向同步。写入RTC需经过2个 `clk_rtc`时钟周期方可生效，此外还涉及 `clk_sys`时钟域。尤其在参考信号频率较低时（例如1Hz），应予以考虑。

4.8.5. 程序员模型

有三项设置任务：

- 设置1秒参考信号
- 设置时钟
- 设置闹钟

4.8.5.1. 配置1秒参考时钟：

选择 `clk_rtc` 的时钟源。该操作在RTC寄存器之外完成（参见第4.8.4节）。

SDK： https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第22至39行

```

22 void rtc_init(void) {
23     // 获取clk_rtc频率并确认其正在运行
24     uint rtc_freq = clock_get_hz(clk_rtc);
25     assert(rtc_freq != 0);
26
27     // 确认clk_rtc运行后解除rtc复位
28     reset_unreset_block_num_wait_blocking(RESET_RTC);
29
30     // 配置1秒分频器。
31     // 若rtc_freq为400，则clkdiv_m1应为399
32     rtc_freq -= 1;
33
34     // 检查频率是否未超过可分配最大值
35     assert(rtc_freq <= RTC_CLKDIV_M1_BITS);
36
37     // 写入分频值
38     rtc_hw->clkdiv_m1 = rtc_freq;
39 }
```

4.8.5.2. 设置时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第54至85行

```

54     bool rtc_set_datetime(const datetime_t *t) {
55         if (!valid_datetime(t)) {
56             return false;
57         }
58
59         // 禁用RTC
60         rtc_hw->ctrl = 0;
61         // 等待其仍处于活动状态
62         while (rtc_running()) {
63             tight_loop_contents();
64         }
65
66         // 写入设置寄存器
67         rtc_hw->setup_0 = (((uint32_t)t->year) << RTC_SETUP_0_YEAR_LSB) |
68             (((uint32_t)t->month) << RTC_SETUP_0_MONTH_LSB) |
69             (((uint32_t)t->day) << RTC_SETUP_0_DAY_LSB);
70         rtc_hw->setup_1 = (((uint32_t)t->dotw) << RTC_SETUP_1_DOTW_LSB) |
71             (((uint32_t)t->hour) << RTC_SETUP_1_HOUR_LSB) |
72             (((uint32_t)t->min) << RTC_SETUP_1_MIN_LSB) |
73             (((uint32_t)t->sec) << RTC_SETUP_1_SEC_LSB);
74
75         // 将设置值加载至 rtc 时钟域
76         rtc_hw->ctrl = RTC_CTRL_LOAD_BITS;
77
78         // 启用 RTC 并等待其正常运行
79         rtc_hw->ctrl = RTC_CTRL_RTC_ENABLE_BITS;
80         while (!rtc_running()) {
81             tight_loop_contents();
82         }
83
84         return true;
85     }

```

① 注意

在 RTC 运行期间，当前时间可以被更改。写入所需数值后，在 CTRL 寄存器中设置 LOAD 位。

4.8.5.3. 读取当前时间

RTC 时间存储于两个 32 位寄存器中。为确保数值一致，应先读取 RTC_0，再读取 RTC_1。读取 RTC_0 会锁存 RTC_1 的值。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第87至106行

```

87     bool rtc_get_datetime(datetime_t *t) {
88         // 确保 RTC 正常运行
89         if (!rtc_running()) {
90             return false;
91         }
92
93         // 注意：RTC_0 应先于 RTC_1 读取
94         uint32_t rtc_0 = rtc_hw->rtc_0;
95         uint32_t rtc_1 = rtc_hw->rtc_1;

```

```

96
97     t->dotw  = (int8_t) (((rtc_0 & RTC_RTC_0_DOTW_BITS ) >> RTC_RTC_0_DOTW_LSB);
98     t->hour  = (int8_t) (((rtc_0 & RTC_RTC_0_HOUR_BITS ) >> RTC_RTC_0_HOUR_LSB);
99     t->min   = (int8_t) (((rtc_0 & RTC_RTC_0_MIN_BITS ) >> RTC_RTC_0_MIN_LSB);
100    t->sec   = (int8_t) (((rtc_0 & RTC_RTC_0_SEC_BITS ) >> RTC_RTC_0_SEC_LSB);
101    t->year  = (int16_t) (((rtc_1 & RTC_RTC_1_YEAR_BITS ) >> RTC_RTC_1_YEAR_LSB));
102    t->month = (int8_t) (((rtc_1 & RTC_RTC_1_MONTH_BITS) >> RTC_RTC_1_MONTH_LSB));
103    t->day   = (int8_t) (((rtc_1 & RTC_RTC_1_DAY_BITS ) >> RTC_RTC_1_DAY_LSB));
104
105    return true;
106 }

```

4.8.5.4. 配置闹钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第146至182行

```

146 void rtc_set_alarm(const datetime_t *t, rtc_callback_t user_callback) {
147     rtc_disable_alarm();
148
149     // 仅当值不为 -1 时加入设置
150     rtc_hw->irq_setup_0 = ((t->year < 0) ? 0 : (((uint32_t)t->year) <<
151                                         RTC_IRQ_SETUP_0_YEAR_LSB)) |
152                                         (((t->month < 0) ? 0 : (((uint32_t)t->month) <<
153                                         RTC_IRQ_SETUP_0_MONTH_LSB)) |
154                                         (((t->day < 0) ? 0 : (((uint32_t)t->day) <<
155                                         RTC_IRQ_SETUP_0_DAY_LSB)));
156     rtc_hw->irq_setup_1 = ((t->dotw < 0) ? 0 : (((uint32_t)t->dotw) <<
157                                         RTC_IRQ_SETUP_1_DOTW_LSB)) |
158                                         (((t->hour < 0) ? 0 : (((uint32_t)t->hour) <<
159                                         RTC_IRQ_SETUP_1_HOUR_LSB)) |
160                                         (((t->min < 0) ? 0 : (((uint32_t)t->min) <<
161                                         RTC_IRQ_SETUP_1_MIN_LSB)) |
162                                         (((t->sec < 0) ? 0 : (((uint32_t)t->sec) <<
163                                         RTC_IRQ_SETUP_1_SEC_LSB)));
164
165     // 设置我们关注项目的匹配使能位
166     if (t->year >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_YEAR_ENA_BITS);
167     if (t->month >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_MONTH_ENA_BITS);
168     if (t->day >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_DAY_ENA_BITS);
169     if (t->dotw >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_DOTW_ENA_BITS);
170     if (t->hour >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_HOUR_ENA_BITS);
171     if (t->min >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_MIN_ENA_BITS);
172     if (t->sec >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_SEC_ENA_BITS);
173
174     // 它会重复吗? 即我们是否不匹配任何位
175     _alarm_repeats = rtc_alarm_repeats(t);
176
177     // 存储稍后可调用的函数指针
178     _callback = user_callback;
179
180     irq_set_exclusive_handler(RTC_IRQ, rtc_irq_handler);
181
182     // 启用外围设备的中断 (IRQ)
183     rtc_hw->inte = RTC_INTE_RTC_BITS;
184
185     // 启用处理器中的 IRQ
186     irq_set_enabled(RTC_IRQ, true);
187
188     rtc_enable_alarm();

```

182 }

i 注意

通过在设置闹钟时使用较少的启用位，可以创建周期性闹钟。例如，如果只匹配秒数且秒设置为 54，则当秒数为 54 时，闹钟中断将每分钟触发一次。

4.8.5.5. 与休眠模式的交互

RP2040 支持两个节能等级：

- 休眠模式，处理器处于休眠状态，芯片中未使用的时钟停止（参见第 2.15.3.5 节）
- 休眠模式，芯片中所有时钟均已停止

RTC 可从这两种模式中唤醒芯片。在休眠模式下，RP2040 可配置为仅运行 `clk_rtc`（一种低速 RTC 参考时钟）及少量允许处理器唤醒的逻辑。当 RTC 闹钟中断触发时，处理器将从休眠模式唤醒。详情请参见第 2.11.5.1 节。

从休眠模式唤醒芯片的方法：

- RTC 必须配置为使用外部参考时钟（由 GPIO 引脚提供）
- 设置 RTC 以使用外部参考时钟运行
- 如果处理器当前由 PLL 供电，需切换为 XOSC/ROSC 供电
- 关闭所有 PLL
- 配置 RTC 为预定的唤醒时间（单次或周期性）
- （可选）关闭大部分存储器电源
- 进入休眠模式（详见第 2.16 节、第 2.17 节及第 2.11.5.2 节）

4.8.6. 寄存器列表

RTC 寄存器起始地址为 `0x4005c000`（在 SDK 中定义为 `RTC_BASE`）。

表 552. RTC 寄存器列表

偏移量	名称	说明
0x00	<code>CLKDIV_M1</code>	1 秒计数器的分频器值减 1。RTC 未启用时，可以安全修改该值。
0x04	<code>SETUP_0</code>	RTC 配置寄存器 0
0x08	<code>SETUP_1</code>	RTC 设置寄存器 1
0x0c	<code>CTRL</code>	RTC 控制与状态
0x10	<code>IRQ_SETUP_0</code>	中断设置寄存器 0
0x14	<code>IRQ_SETUP_1</code>	中断设置寄存器 1
0x18	<code>RTC_1</code>	RTC 寄存器 1
0x1c	<code>RTC_0</code>	RTC 寄存器 0 使用 RTC 1 之前请先阅读本说明！
0x20	<code>INTR</code>	原始中断
0x24	<code>INTE</code>	中断使能

偏移量	名称	说明
0x28	INTF	中断强制
0x2c	INTS	掩码及强制后的中断状态

RTC: CLKDIV_M1 寄存器

偏移: 0x00

表 553. CLKDIV_M1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	1秒计数器的分频器值减1。 RTC未启用时，可以安全修改该值。	读写	0x0000

RTC: SETUP_0 寄存器

偏移: 0x04

描述

RTC配置寄存器0

表 554. SETUP_0
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	YEAR : 年	读写	0x000
11:8	MONTH : 月 (1..12)	读写	0x0
7:5	保留。	-	-
4:0	DAY : 月内日期 (1..31)	读写	0x00

RTC: SETUP_1 寄存器

偏移: 0x08

描述

RTC 设置寄存器 1

表 555. SETUP_1
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26:24	DOTW : 星期几，1-星期一...0-星期日，ISO 8601 模 7	读写	0x0
23:21	保留。	-	-
20:16	HOUR : 小时	读写	0x00
15:14	保留。	-	-
13:8	MIN : 分钟	读写	0x00
7:6	保留。	-	-
5:0	SEC : 秒	读写	0x00

RTC: CTRL 寄存器

偏移: 0x0c

说明

RTC 控制与状态

表 556. CTRL
寄存器

位	描述	类型	复位值
31:9	保留。	-	-
8	FORCE_NOTLEAPYEAR : 若设置，则强制非闰年。 适用于能被100整除但不能被400整除的年份	读写	0x0
7:5	保留。	-	-
4	LOAD : 载入 RTC	SC	0x0
3:2	保留。	-	-
1	RTC_ACTIVE : RTC 已启用（运行中）	只读	-
0	RTC_ENABLE : 启用 RTC	读写	0x0

RTC: IRQ_SETUP_0 寄存器

偏移: 0x10

描述

中断设置寄存器 0

表 557.
IRQ_SETUP_0 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29	MATCH_ACTIVE	只读	-
28	MATCH_ENA : 全局匹配使能。启用此项时请勿更改其他任何值	读写	0x0
27	保留。	-	-
26	YEAR_ENA : 启用年匹配	读写	0x0
25	MONTH_ENA : 启用月匹配	读写	0x0
24	DAY_ENA : 启用日匹配	读写	0x0
23:12	YEAR : 年	读写	0x000
11:8	MONTH : 月 (1..12)	读写	0x0
7:5	保留。	-	-
4:0	DAY : 月内日期 (1..31)	读写	0x00

RTC: IRQ_SETUP_1 寄存器

偏移量: 0x14

描述

中断设置寄存器 1

表 558.
IRQ_SETUP_1 寄存器

位	描述	类型	复位值
31	DOTW_ENA : 启用星期匹配	读写	0x0
30	HOUR_ENA : 启用小时匹配	读写	0x0
29	MIN_ENA : 启用分钟匹配	读写	0x0
28	SEC_ENA : 启用秒匹配	读写	0x0

位	描述	类型	复位值
27	保留。	-	-
26:24	DOTW: 星期	读写	0x0
23:21	保留。	-	-
20:16	HOUR: 小时	读写	0x00
15:14	保留。	-	-
13:8	MIN: 分钟	读写	0x00
7:6	保留。	-	-
5:0	SEC: 秒	读写	0x00

RTC: RTC_1 寄存器

偏移: 0x18

描述

RTC 寄存器 1

表 559。RTC_1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	YEAR: 年	只读	-
11:8	MONTH: 月 (1..12)	只读	-
7:5	保留。	-	-
4:0	DAY: 月内日期 (1..31)	只读	-

RTC: RTC_0 寄存器

偏移量: 0x1c

描述

RTC 寄存器 0

使用 RTC 1 之前请先阅读本说明!

表 560。RTC_0
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26:24	DOTW: 星期	RF	-
23:21	保留。	-	-
20:16	HOUR: 小时	RF	-
15:14	保留。	-	-
13:8	MIN: 分钟	RF	-
7:6	保留。	-	-
5:0	SEC: 秒	RF	-

RTC: INTR 寄存器

偏移量: 0x20

说明

原始中断

表 561。INTR 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	只读	0x0

RTC：INTE 寄存器

偏移: 0x24

说明

中断使能

表 562。INTE 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	读写	0x0

RTC：INTF 寄存器

偏移: 0x28

说明

中断强制

表 563。INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	读写	0x0

RTC：INTS 寄存器

偏移量: 0x2c

描述

掩码及强制后的中断状态

表 564。INTS 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	只读	0x0

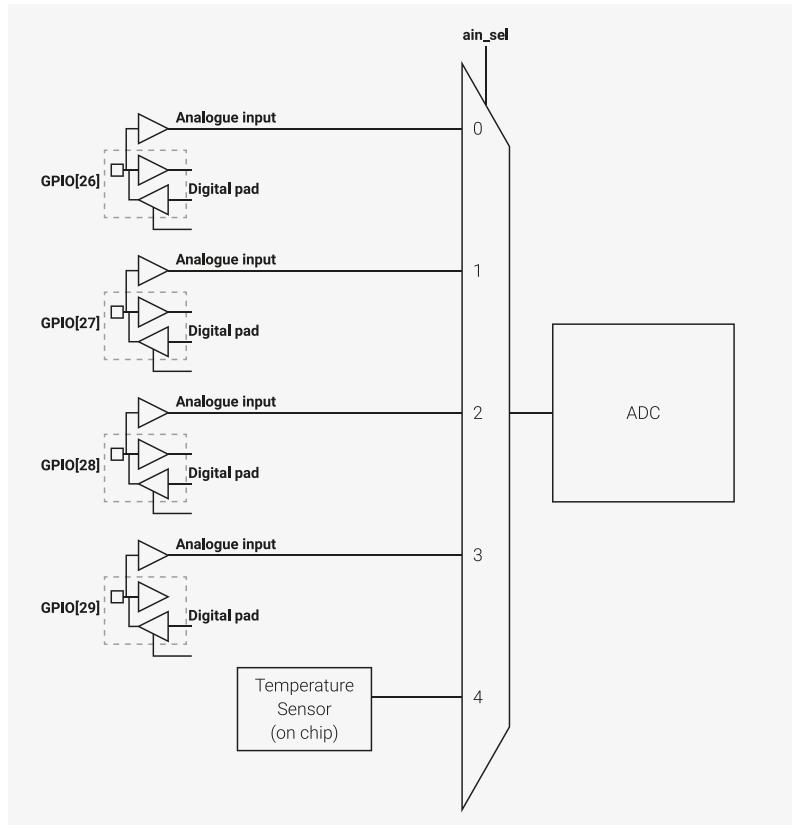
4.9. 模数转换器 (ADC) 及温度传感器

RP2040 内置模拟数字转换器 (ADC)，具备以下特性：

- 逐次逼近寄存器 (SAR) ADC (参见第 4.9.2 节)
- 500ksps (采用独立 48MHz 时钟)
- 12 位，具有 8.7 有效位数 (ENOB) (参见第 4.9.3 节)
- 五路输入复用器：
 - 四个输入通过封装引脚共享 GPIO[29:26]

- 一路输入专用用于内部温度传感器（参见第 4.9.5 节）
- 八元素采样接收 FIFO
- 中断生成
- DMA 接口（参见第 4.9.2.5 节）

图 114. ADC
连接示意图



注意

使用与 GPIO 引脚共享的 ADC 输入时，必须通过将该引脚的垫控寄存器中的IE 置低和OD 置高以禁用该引脚的数字功能。详见第2.19.6.3节，“焊盘控制 - 用户银行”。最大ADC输入电压由数字IO电源电压（IOVDD）决定，而非ADC电源电压（ADC_AVDD）。例如，若IOVDD供电为1.8V，即使ADC_AVDD供电为3.3V，ADC输入电压也不得超过1.8V。超过IOVDD的电压将通过ESD保护二极管产生泄漏电流。详见第5.5.3节，“引脚规格”。

4.9.1. ADC 控制器

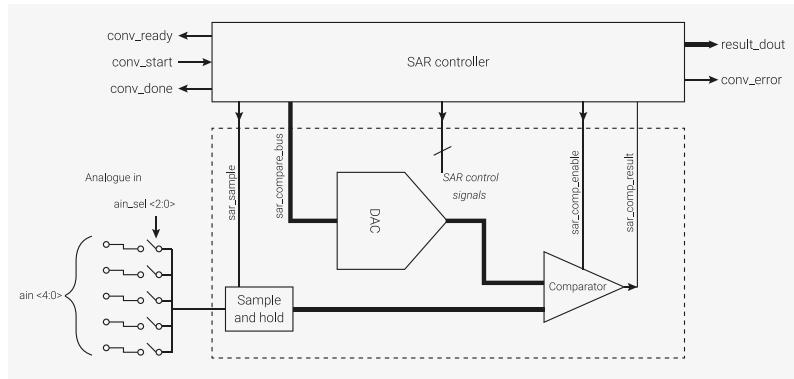
数字控制器负责管理RP2040 ADC的具体操作，并提供以下附加功能：

- 单次采集或自由运行采样模式
- 带DMA接口的采样FIFO
- 用于设定自由运行采样率的定时器（16位整数，8位小数）
- 自由运行采样模式下的多通道轮询采样
- 自由运行采样模式下，可选择右移至8位，以便通过DMA传输至系统内存中的字节缓冲区

4.9.2. SAR型ADC

SAR ADC（逐次逼近寄存器模数转换器）是数字控制器与模拟电路的组合，如图115所示。

图115。SAR ADC
框图



ADC需要48MHz时钟（`clk_adc`），该时钟可由USB PLL提供。采样耗时为96个时钟周期 ($96 \times 1/48\text{MHz} = 2\mu\text{s}$ 每样本 (500ksps))。启用ADC之前，时钟必须正确配置。

一旦ADC模块获得时钟且复位解除，向CS.EN写入1将启动ADC模拟硬件的短暂内部上电序列。经过数个时钟周期后，CS.READY信号将置高，表示ADC已准备开始首次转换。

可通过清除CS.EN随时禁用ADC，以节省功耗。CS.EN不启用温度传感器偏置电源（见第4.9.5节），此项由独立控制。

ADC输入为电容性，采样时会在输入端施加约1pF电容（ADC外部如封装和PCB走线还会增加额外电容）。即使以500ksps采样，有效阻抗仍高于100kΩ，对于直流测量通常无需缓冲。

4.9.2.1 单次采样

向CS.START_ONCE写入1将立即启动新的一次转换。CS.READY将置低，表示转换正在进行中。经过96个`clk_adc`周期后，CS.READY将置高。12位转换结果可在RESULT中获得。

采样的ADC输入可通过在转换开始前任意时刻写入CS.AINSEL进行选择。当AINSEL值为0~3时，选择GPIO 26~29作为ADC输入。当AINSEL值为4时，选择内部温度传感器。

注意

切换AINSEL时无需等待稳定时间。

4.9.2.2. 自由运行采样

当CS.START_MANY置位时，ADC将自动以固定间隔启动新的转换。最新的转换结果始终可在RESULT中读取，但对于通过IRQ或DMA进行的采样流，必须启用ADC FIFO（参见第4.9.2.4节）。

默认情况下(DIV=0)，新转换在前一次转换完成后立即开始，因此每96个时钟周期产生一个新样本。在48MHz时钟频率下，相当于500ksps。

将DIV.INT设置为某个正值n会使ADC每n+1个周期触发一次转换，但若当前转换尚未完成，ADC将忽略该设置，因此n通常需要满足 ≥ 96 。例如，在48MHz时钟频率下，将DIV.INT设置为47999，ADC可按1ksps运行。

定时器支持分数倍率分频（一阶三角Σ-Δ调制）。当将DIV.FRAC设置为非零值时，

ADC 将以平均每
介于 INT + 1 和 INT + 2.

$$1 + \text{INT} + \frac{\text{FRAC}}{256}$$

个周期启动一次新的转换，通过更改采样间隔

4.9.2.3. 多输入采样

[CS.RROBIN 允许 ADC 在自由运行采样过程中以轮询方式采样多个输入。](#)

RROBIN 寄存器中的每一位对应于 CS.AINSEL 的五个可能值之一。当 ADC 完成转换后，CS.AINSEL 会自动切换至 RROBIN 中对应位被设置的下一个输入。

将全零写入 CS.RROBIN 即可禁用轮询采样功能。

例如，若AINSEL 初始设为 0，且RROBIN 设置为 0x06（第 1 和第 2 位被设置），ADC 会按以下顺序采样通道：

1. 通道 0
2. 通道 1
3. 通道 2
4. 通道 1
5. 通道 2
6. 通道 1...

i 注意

AINSEL 的初始值无需与 RROBIN 中被置位的位相对应。

4.9.2.4. 采样 FIFO

ADC 采样值可以直接从 RESULT 寄存器读取，或存储在本地包含 8 个条目的 FIFO 中，再从 FIFO 中读取。FIFO 的操作由 FCS 寄存器控制。

如果设置了 FCS.EN，每次 ADC 转换结果均写入 FIFO。当 ADC 的 IRQ 或 DREQ 信号发出通知时，软件中断处理程序或 RP2040 DMA 可从 FIFO 中读取该采样。另外，软件也可通过轮询 FCS 的状态位以等待每个采样数据的可用。

若转换完成时 FIFO 已满，将设置粘滞错误标志 FCS.OVER。此事件不会更改当前 FIFO 内容，但所有在 FIFO 已满时完成的转换采样将被丢失。

有两个标志控制ADC向FIFO写入的数据：

- [FCS.SHIFT 会将 FIFO 数据右移至八位大小（即 FIFO 位 7:0 对应转换结果的位 11:4）](#)。此举适用于对内存中的字节缓冲区进行 8 位 DMA 传输，允许更深的捕获缓冲区，但会以牺牲部分精度为代价。
- [FCS.ERR 会设置每个 FIFO 值的 FIFO.ERR 标志，表示发生了转换错误，即 SAR 未能收敛（详见下文）。](#)

⚠ 注意

转换错误会产生未定义结果，应丢弃相应的采样数据。这些标志表明一个或多个位的比较未能在允许时间内完成。通常这是由比较器亚稳态引起，即输入信号越接近比较器阈值，作出决策所需时间越长。

比较器的高增益降低了未能作出决策的概率。

4.9.2.5. DMA

RP2040 DMA（第2.5节）可通过对FIFO寄存器进行普通内存映射读取，从样本FIFO获取ADC采样，该读取由 [ADC_DREQ](#) 系统数据请求信号控制。须考虑以下事项：

- 必须启用样本FIFO（FCS.EN），以允许样本写入其中；FIFO默认处于禁用状态，以防止ADC用于单次转换时 FIFO被意外填满。
- 必须通过FCS.DREQ_EN启用ADC数据请求握手（DREQ）。
- 用于传输的DMA通道必须选择 [DREQ_ADC](#) 数据请求信号（第2.5.3.1节）。
- DREQ断言阈值（FCS.THRESH）应设置为1，以确保FIFO中只要存在单个样本，DMA即刻开始传输。请注意，该阈值也是IRQ断言所用，非DMA应用场景可能会优先选择更高阈值以减少中断频率。
- 如果DMA传输大小设置为8位，即DMA传输至内存中的字节数组，则FCS.SHIFT必须设置，以预先将FIFO采样数据左移至8位有效位。
- 若需采样多个输入通道，CS.RROBIN包含一个5位掩码，标识这些通道（4个外部输入及温度传感器）。此外，CS.A_INSEL须选择首次采样的通道。
- 应在启动ADC之前配置ADC采样率（见第4.9.2.2节）。

ADC配置完成后，应先启动DMA通道，再通过CS.START_MANY启动ADC转换。DMA完成后，可停止ADC，或立即开始新的DMA传输。清除CS.START_MANY以停止ADC后，软件应轮询CS.READY以确认最后转换完成，并清空FIFO中残留的样本。

4.9.2.6. 中断

当 FIFO 水位达到可配置阈值 FCS.THRESH 时，可触发中断。中断输出须通过 INTEN 使能。

状态可通过 INTS 读取。中断通过将 FIFO 排空至低于 FCS.THRESH 水位来清除。

4.9.2.7. 电源

ADC 电源被独立分接至专用引脚以便进行噪声滤波。

4.9.3. ADC 有效位数 (ENOB)

对 ADC 进行了性能表征，并测量了 ADC 的 ENOB。测试在室温下对不同硅片批次实施，涵盖 3 个典型 ([tt](#))、3 个快速 ([ff](#)) 及 3 个慢速 ([ss](#)) 工艺边界 RP2040 器件。

表 566 中的典型值、最小值和最大值反映了用于测试的硅片特性。

表 565。 测试中使用的参数。

参数	数值
采样率	250ksps

参数	数值
FFT 窗函数	5 项 Blackman-Harris 窗
FFT 频段	4,096
FFT 平均值	无
输入电平最小值	1
输入电平最大值	4,094
输入频率	997Hz

应注意，THD通常使用前5或6个谐波进行计算。然而，由于INL/DNL误差（参见第4.9.4节）产生的峰值多于此数量，故采用前30个峰值。这导致THD值略微偏高，但更能反映实际情况。

表566。不同部件的测试结果（快速、慢速及典型）。

	最小值	典型值	最大值
THD ¹	-55.6dB	55dB	-54.4dB
信噪比	60.9dB	61.5dB	62.0dB
SFDR	59.2dB	59.9dB	60.5dB
SINAD	53.6dB	54.0dB	54.6dB
ENOB	8.6	8.7	8.8

¹由于INL产生大量谐波，因此采用了最高的30个峰值。这与传统的THD计算方法不同。

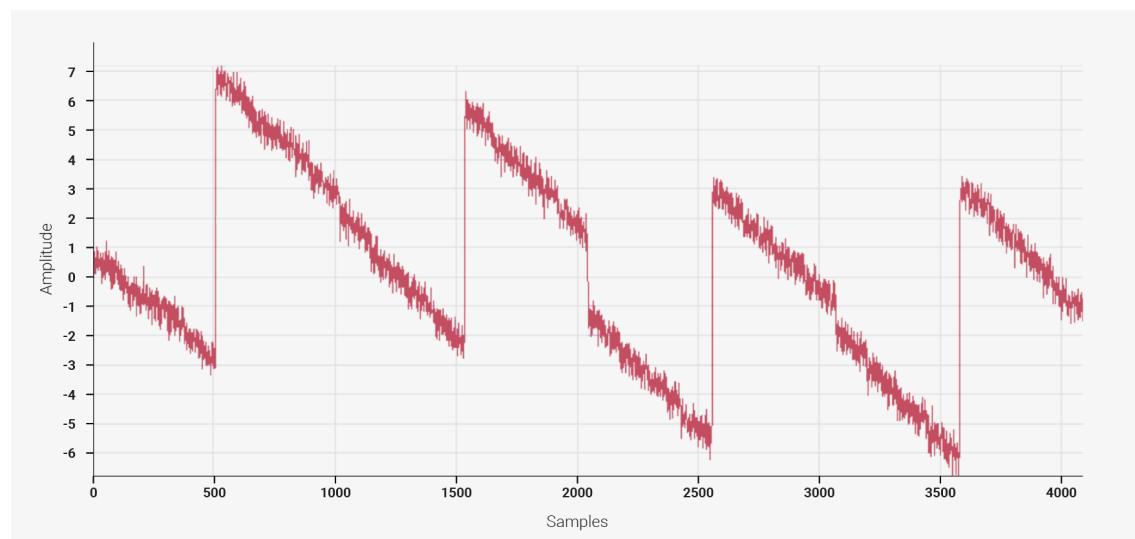
！重要

测试采用带有低噪声板载电压基准的板卡进行，因为在对ADC进行特性测试时，确保无其他噪声源影响测量至关重要。

4.9.4. 积分非线性 (INL) 与差分非线性 (DNL)

积分非线性 (INL) 与微分非线性 (DNL) 用于衡量ADC对输入信号量化误差的大小。理想 ADC 中，输入到输出的传输函数应在线性量化传递模拟输入信号与数字输出信号之间。RP2040 ADC中每个二进制结果对应的INL值如图116所示，表明误差呈锯齿形而非预期曲线。

图116。ATE
设备测试结果
INL (RP2040)。

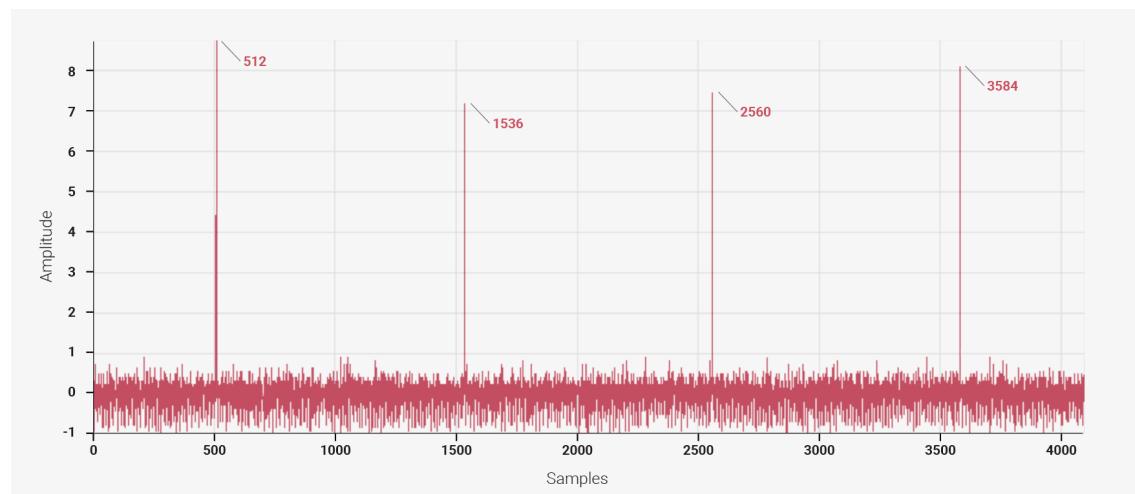


名义上，ADC从一个数字值移动到下一个数字值，通俗地称为“无缺码”。

然而，若ADC跳过某个数值区间，将导致差分非线性（DNL）误差出现尖峰。此类误差通常仅在特定码值处出现，系由ADC设计所致。

RP2040的ADC具有基本平坦且低于1 LSB的DNL，但在512、1536、2560及3584四个数值处，其DNL误差达到峰值，见图117。

图117。ATE
设备测试结果
DNL (RP2040)。



INL与DNL误差源于ADC内部部分电容的缩放比例偏差。因这些电容数值极小（仅数十飞法拉），芯片仿真结果在此微小数值段可能与实际有细微偏差。如果这些电容器匹配正确，ADC的性能本可更优。

这些INL和DNL误差将依据具体使用情况，在一定程度上限制ADC的性能（参见勘误 RP2040-E11）。

4.9.5. 温度传感器

温度传感器测量一个偏置的双极性二极管的V_{be}电压，该二极管连接至第五ADC通道（AINSEL=4）。通常，V_{be}在27摄氏度时为0.706V，斜率为每摄氏度-1.721mV。因此，温度可近似计算如下：

$$T = 27 - (\text{ADC_voltage} - 0.706)/0.001721$$

鉴于V_{be}及其斜率会随温度及器件差异变化，若需精确测量，用户可能需要进行校准。

温度传感器的偏置源必须通过 CS_TS_EN 使能后方可使用，此操作会使 ADC_AVDD 电流增加约 40 μ A。

i 注意

板载温度传感器对基准电压的误差极为敏感。若ADC返回值为891，则对应温度为20.1°C。然而，若参考电压比3.3V低1%，则相同的891读数对应的温度为24.3°C。参考电压仅1%的微小变化，即引起温度变化超过4°C。因此，若您希望提高内部温度传感器的精度，建议考虑添加外部参考电压。

i 注意

INL误差（参见第4.9.4节）不属于ADC的可用温度范围。

4.9.6. 寄存器列表

ADC寄存器的基址起始为 **0x4004c000**（在SDK中定义为ADC_BASE）。

表 567. ADC 寄存器列表

偏移量	名称	说明
0x00	CS	ADC控制与状态
0x04	RESULT	最近一次ADC转换的结果
0x08	FCS	FIFO控制与状态
0x0c	FIFO	转换结果 FIFO
0x10	DIV	时钟分频器。若非零，CS_START_MANY 会启动转换 以固定间隔而非连续进行。 任一字段写入时，分频器将被重置。 总周期为 $1 + INT + FRAC / 256$
0x14	INTR	原始中断
0x18	INTE	中断使能
0x1c	INTF	中断强制
0x20	INTS	掩码及强制后的中断状态

ADC：CS 寄存器

偏移: 0x00

描述

ADC控制与状态

表 568: CS 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:16	RROBIN : 轮询采样，每通道 1 位。将所有位设置为 0 以禁用。 否则，ADC 将以轮询方式依次转换每个启用的通道。 首次采样通道为当前AINSEL指定的通道。 AINSEL会在每次转换后自动更新为新选通道。	读写	0x00
15	保留。	-	-

位	描述	类型	复位值
14:12	AINSEL : 选择模拟多路复用输入。在轮询模式下自动更新。	读写	0x0
11	保留。	-	-
10	ERR_STICKY : 先前某次 ADC 转换发生错误，写 1 清除。	WC	0x0
9	ERR : 最近一次 ADC 转换发生错误；结果未定义或含噪声。	只读	0x0
8	READY : 当ADC准备开始新转换时置1。表示先前的转换已完成。 转换进行中时置0。	只读	0x0
7:4	保留。	-	-
3	START_MANY : 当该位为1时，连续执行转换。前一次转换结束后立即开始新转换。	读写	0x0
2	START_ONCE : 启动单次转换。自动清零。若start_many置位则忽略此位。	SC	0x0
1	TS_EN : 温度传感器上电。1表示启用，0表示禁用。	读写	0x0
0	EN : ADC上电并使能时钟。 1表示启用，0表示禁用。	读写	0x0

ADC: RESULT寄存器

偏移: 0x04

表569. RESULT
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	最近一次ADC转换的结果	只读	0x000

ADC: FCS寄存器

偏移: 0x08

描述

FIFO控制与状态

表570. FCS
寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	THRESH : 当电平 \geq 阈值时，DREQ/IRQ被触发	读写	0x0
23:20	保留。	-	-
19:16	LEVEL : FIFO中当前等待的转换结果数量	只读	0x0
15:12	保留。	-	-
11	OVER : FIFO溢出时置1。写1以清除该状态。	WC	0x0
10	UNDER : FIFO欠流时置1。写1以清除该状态。	WC	0x0
9	满	只读	0x0
8	空	只读	0x0
7:4	保留。	-	-

位	描述	类型	复位值
3	DREQ_EN : 若为1，当FIFO中含有数据时，断言DMA请求	读写	0x0
2	ERR : 若为1，转换错误位将与结果一同出现在FIFO中	读写	0x0
1	SHIFT : 若为1，FIFO结果向右移位，变为一字节大小。启用对字节缓冲区的DMA。	读写	0x0
0	EN : 若为1，每次转换后将结果写入FIFO。	读写	0x0

ADC: FIFO寄存器

偏移: 0x0c

描述

转换结果 FIFO

表571. FIFO
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	ERR : 若此特定采样发生转换错误，则为1。若采样发生移位，则保持在相同位置。	RF	-
14:12	保留。	-	-
11:0	值	RF	-

ADC: DIV寄存器

偏移: 0x10

说明

时钟分频器。若非零，CS_START_MANY将按固定间隔启动转换，而非连续启动。

任一字段写入时，分频器将被重置。

总周期为 $1 + \text{INT} + \text{FRAC} / 256$

表572. DIV
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:8	INT : 时钟分频器的整数部分。	读写	0x0000
7:0	FRAC : 时钟分频器的小数部分。一级 Δ - Σ 调制。	读写	0x00

ADC: INTR寄存器

偏移: 0x14

描述

原始中断

表 573. INTR 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	只读	0x0

ADC: INTE 寄存器

偏移: 0x18

描述

中断使能

表 574. INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	读写	0x0

ADC: INTF 寄存器

偏移: 0x1c

描述

中断强制

表 575. INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	读写	0x0

ADC: INTS 寄存器

偏移: 0x20

说明

掩码及强制后的中断状态

表 576. INTS 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	只读	0x0

4.10. 串行同步接口 (SSI)

Synopsys文档

Synopsys专有，经过授权使用。

RP2040 配备同步串行接口 (SSI) 控制器，该控制器位于 QSPI 引脚，用于与外部 Flash 设备通信。该 SSI 控制器属于 XIP 模块的一部分。

SSI 控制器基于 Synopsys DW_apb_ssi IP (版本 4.01a) 配置而成。

4.10.1. 概述

为使 DW_apb_ssi 能连接至串行主设备或串行从设备外设，外设必须至少具备以下任一接口：

摩托罗拉串行外围接口（SPI）

摩托罗拉公司提出的一种四线全双工串行协议。串行时钟的相位及极性共有四种可能组合。时钟相位（SCPH）决定串行传输是从从设备选择信号的下降沿开始，还是从串行时钟的第一个边沿开始。当 DW_apb_ssi 处于空闲或禁用状态时，从设备选择线保持高电平。

德州仪器串行协议（SSP）

一种四线全双工串行协议。用于 SPI 和 Microwire 协议的从设备选择线兼作 SSP 协议的帧指示线。

国家半导体 Microwire 协议

一种半双工串行协议，通过串行主设备向目标串行从设备传输控制字。

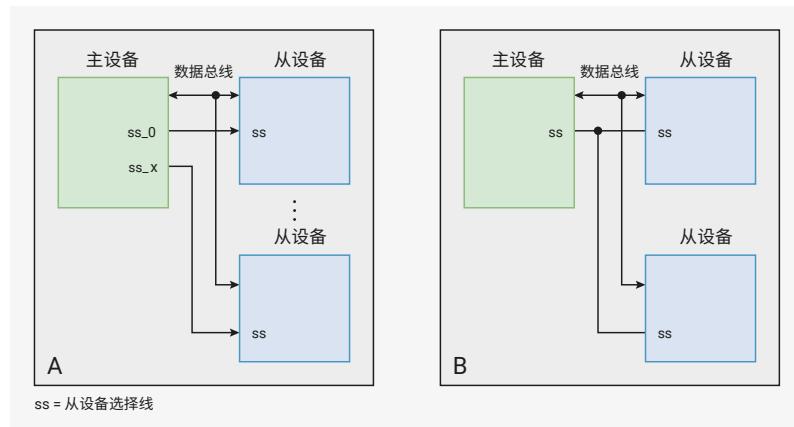
用户可通过控制寄存器 0 (CTRLR0) 中的 FRF (帧格式) 位字段编程，选择所用协议。

DW_apb_ssi 支持的串行协议允许使用硬件或软件方式选择或寻址串行从设备。在硬件实现时，串行从设备通过专用的硬件选择线进行选取。串行主设备产生的选择线数量应等于总线上的串行从设备数量。在数据传输开始之前，串行主设备会使目标串行从设备的选择线有效。该架构如图118所示。

在软件实现时，所有串行从设备的输入选择线应来自串行主设备的单一从设备选择输出。在此模式下，假设串行主设备仅有一个从设备选择输出。如果系统中存在多个串行主设备，则所有主设备的从设备选择输出可通过逻辑与操作生成所有串行从设备的单一从设备选择输入。软件主程序负责控制目标从设备的选择；该架构如图118所示。软件会利用所有从设备中的SSIENR寄存器来控制响应主设备串行传输请求的从设备。

DW_apb_ssi 不对串行从设备的硬件或软件控制进行强制管理。您可以依据图 118 所示配置 DW_apb_ssi，以实现任一方方案。

图 118。
硬件/软件
从设备选择。



4.10.2. 特性

DW_apb_ssi 是一款可配置且可编程的组件，作为全双工主序列接口。主处理器通过 APB 接口访问 DW_apb_ssi 的数据、控制及状态信息。DW_apb_ssi 还通过 DMA 控制器实现大批量数据传输。

DW_apb_ssi 配置为串行主设备。DW_apb_ssi 可通过以下任何接口连接至任意串行从外设设备：

- 摩托罗拉串行外围接口（SPI）
- 德州仪器串行协议（SSP）
- 国家半导体 Microwire 协议

在 RP2040 上，DW_apb_ssi 是闪存执行就地子系统的组成部分（参见第 2.6.3 节），提供与外部 SPI、双 SPI 或四 SPI 闪存设备的通信。

4.10.2.1. 输入/输出连接

SSI 控制器连接至以下引脚：

- `QSPI_SCLK` 连接至输出时钟 `sclk_out`
- `QSPI_SS_N` 连接至片选信号 `ss_o_n`
- `QSPI_SD[3:0]` 连接至数据总线 `txd` 及 `rxn`

IP 上部分引脚被固定为未使用状态：

- `ss_in_n` 被拉高

时钟连接如下：

- `pclk` 及 `sclk` 由 `clk_sys` 驱动

4.10.3. IP 修改

对 Synopsys DW_apb_ssi 硬件进行了以下修改：

1. XIP 访问采用字节交换方式，最低地址字节位于最低有效字节位置
2. 当 `SPI_CTRLR0_INST_L` 为 0 时，XIP 指令字段追加于地址末尾，而非前置于地址起始
3. 寄存器 `DMARDLR` 的复位值由 0 调整为 4。RP2040 上 SSI 与 DMA 的握手依据 RX FIFO 水平是否达到 `DMARDLR`，仅请求单次传输或四次突发传输，故 `DMARDLR` 不宜更改。

这些更改中的第一个允许小端总线主控设备进行混合大小访问，例如 RP2040 的 DMA 或 RP2040 上使用的 Cortex-M0+ 配置。请注意，此更改仅适用于 XIP 访问（RP2040 系统地址范围 `0x10000000` 至 `0x13fffff`），不适用于直接访问 DW_apb_ssi FIFO。直接访问 SSI 时，软件可能需手动进行字节交换，或使用 RP2040 DMA 的字节交换功能。

第二项更改支持在 XIP 地址后发出续传位，从而支持无命令前缀的 XIP 模式（例如 Winbond 设备的 `EBh` Quad I/O 快速读取），以提升性能。例如，以下配置用于对每次访问 XIP 地址窗口时发出标准的 `03h` 串行读取命令：

- `SPI_CTRLR0_INST_L = 8 bits`
- `SPI_CTRLR0_ADDR_L = 24 位`
- `SPI_CTRLR0_XIP_CMD = 0x03`

此操作将首先发出八位命令位（`0x03`），随后发出 24 位地址位，最后时钟输入数据位。用于 `EBh` 四线读取的配置，在闪存进入 XIP 状态后，配置如下：

- `SPI_CTRLR0_INST_L = 0`
- `SPI_CTRLR0_ADDR_L = 32 位`
- `SPI_CTRLR0_XIP_CMD = 0xa0` （适用于 W25Qx 设备的续传码）

每次 XIP 访问时，DW_apb_ssi 将发送 32 个“地址”位，其中包含 RP2040 系统总线最低 24 位地址。

， 并紧随其后发送8位续传码 **0xa0**。 不发送命令前缀。

4.10.3.1. 使用软件进行目标从设备选择的示例

以下示例为伪代码，演示如何使用软件选择目标从设备。

```
1 int main() {
2     disable_all_serial_devices(); ①
3     initialize_mst(ssi_mst_1); ②
4     initialize_slv(ssi_slv_1); ③
5     start_serial_xfer(ssi_mst_1); ④
6 }
```

①此函数将串行总线中每个设备的SSIENR寄存器的SSI_EN位设置为逻辑'0'。

②此函数用于初始化串行传输的主设备；

③此函数初始化目标从设备（本例为从设备1），以进行串行传输；

④此函数通过将传输数据写入主设备的TX FIFO来启动串行传输。用户可通过函数轮询忙碌状态，或使用中断服务程序（ISR）判断串行传输何时完成。

1. Write CTRLR0 to match the required transfer
2. If transfer is receive only write number of frames into CTRLR1
3. Write BAUDR to set the transfer baud rate.
4. Write TXFTLR and RXFTLR to set FIFO threshold levels
5. Write IMR register to set interrupt masks
6. Write SER register bit[0] to logic '1'
7. Write SSIENR register bit[0] to logic '1' to enable the master.

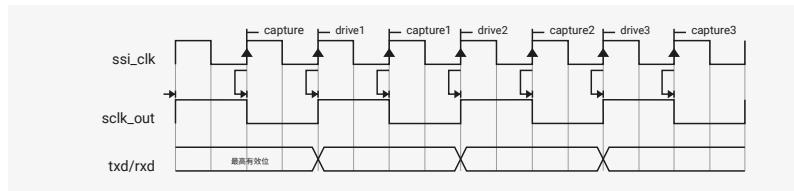
1. Write CTRLR0 to match the required transfer
2. Write TXFTLR and RXFTLR to set FIFO threshold levels
3. Write IMR register to set interrupt masks
4. 将SSIENR寄存器的第0位写为逻辑“1”，以启用从设备。
5. 若从设备需传输数据，则将数据写入TX FIFO。此时从设备已启用，正等待其ss_in_n输入端口的有效电平。请注意，所有其他串行从设备均被禁用（SSI_EN=0），因此不会响应其ss_in_n端口的有效电平。

4.10.4. 时钟比率

位速率时钟（sclk_out）的最大频率为ssi_clk频率的一半。该设计允许移位控制逻辑在sclk_out的一个时钟边沿采集数据，并在相反边沿传输数据。

[图119示意了sclk_out与ssi_clk之间的最大频率比。](#)

图119。最大
sclk_out/ssi_clk 比率。



仅在正在进行有效传输时，sclk_out线路才会切换。其他时间，该线路保持在非活动状态，状态定义依据其所遵循的串行协议。

sclk_out的频率可由以下公式导出：

$$F_{sclk_out} = \frac{F_{ssi_clk}}{SCKDV}$$

SCKDV是可编程寄存器BAUDR中的位域，取值为0至65534之间的任一偶数。若SCKDV为0，则sclk_out被禁用。

4.10.4.1 频率比摘要

bit率时钟（sclk_out）与DW_apb_ssi外设时钟（ssi_clk）之间频率比的限制摘要如下：

- $F_{ssi_clk} >= 2 \times (\text{maximum } F_{sclk_out})$

4.10.5. 发送与接收 FIFO 缓冲区

DW_apb_ssi使用的FIFO缓冲区为内部16级D型触发器。由于串行规范规定串行传输（数据帧）的长度可为4至16/32位，发送和接收FIFO缓冲区的宽度固定为32位。不足32位的数据帧在写入发送FIFO缓冲区时必须右对齐。移位控制逻辑自动将接收FIFO缓冲区中的接收数据右对齐。

FIFO缓冲区中的每个数据项包含单个数据帧。不可能在单个FIFO位置存储多个数据帧；例如，不能在单个FIFO位置存储两个8位数据帧。若需要8位数据帧，串行移位器传输数据时将忽略FIFO项的高位。

注意

当DW_apb_ssi被禁用（SSI_EN = 0）或复位（presetn）时，发送和接收FIFO缓冲区将被清空。（present）。

通过对DW_apb_ssi数据寄存器（DR）的APB写命令加载发送FIFO。数据由移位控制逻辑从发送FIFO弹出（移除）至发送移位寄存器。当FIFO中条目数小于或等于FIFO阈值时，发送FIFO产生FIFO空中断请求（ssi_txe_intr）。阈值由可编程寄存器TXFTLR设置，决定触发中断的FIFO条目数量。该阈值可提前向处理器指示发送FIFO即将空空。若尝试向已满的发送FIFO写入数据，将触发发送FIFO溢出中断（ssi_txo_intr）。

数据由APB读命令从接收FIFO弹出，读取DW_apb_ssi数据寄存器（DR）。接收FIFO由移位控制逻辑从接收移位寄存器加载。当FIFO中条目数大于或等于阈值加1时，接收FIFO产生FIFO满中断请求（ssi_rxf_intr）。该阈值通过可编程寄存器RXFTLR设置，决定触发中断的FIFO条目数量。

阈值允许您提前向处理器指示接收 FIFO 即将满。当接收移位逻辑尝试将数据加载到已满的接收 FIFO 时，会触发接收 FIFO 溢出中断（ssi_rxo_intr）。然而，此时新接收的数据将丢失。当满足以下条件时，会触发接收 FIFO 欠载中断（ssi_rxu_intr）：

尝试从空的接收 FIFO 读取数据时。此行为提醒处理器读取的数据无效。

[表 577 说明了不同发送 FIFO 阈值的含义。](#)

表 577。发送 FIFO 阈值 (TFT)
解码值

TFT 值	描述
0000_0000	当发送 FIFO 中无数据项时, ssi_txe_intr 被触发
0000_0001	当发送 FIFO 中的数据项少于或等于一项时, ssi_txe_intr 被触发
0000_0010	当发送 FIFO 中的数据项少于或等于两项时, ssi_txe_intr 被触发
...	...
0000_1101	当发送 FIFO 中的数据项少于或等于十三项时, ssi_txe_intr 被触发
0000_1110	当发送 FIFO 中存在不超过 14 个数据条目时, ssi_txe_intr 被置位
0000_1111	当发送 FIFO 中存在不超过 15 个数据条目时, ssi_txe_intr 被置位

[表 578 说明了不同接收 FIFO 阈值的含义。](#)

表 578。接收 FIFO 阈值 (RFT)
解码值

RFT 值	描述
0000_0000	当接收 FIFO 中存在一个或多个数据条目时, ssi_rxf_intr 被置位
0000_0001	当接收 FIFO 中存在两个或多个数据条目时, ssi_rxf_intr 被置位
0000_0010	当接收 FIFO 中存在三个或多个数据条目时, ssi_rxf_intr 被置位
...	...
0000_1101	当接收 FIFO 中存在 14 个或更多数据条目时, ssi_rxf_intr 被置位
0000_1110	当接收 FIFO 中存在 15 个或更多数据条目时, ssi_rxf_intr 被置位
0000_1111	当接收 FIFO 中存在 16 个数据条目时, ssi_rxf_intr 被置位

4.10.6. 32 位帧尺寸支持

该 IP 配置的数据帧大小最大可编程值为 32 位。因此，具备以下特性：

- dfs_32 (CTRLR0[20:16]) 有效，包含数据帧大小的值。该新寄存器字段的取值范围为0至31。dfs (CTRLR0[3:0]) 无效，写入此寄存器无任何效果。
- 接收和发送FIFO的宽度均为32位。
- 数据寄存器的全部32位均为有效。

4.10.7. SSI 中断

DW_apb_ssi支持组合及单独中断请求，且均可屏蔽。组合中断请求为所有其他DW_apb_ssi中断屏蔽后的逻辑或结果。仅组合中断请求被路由至中断控制器。所有DW_apb_ssi中断均为电平中断，且高电平有效。

DW_apb_ssi中断说明如下：

发送FIFO中断 (ssi_txe_intr)

当发送FIFO的填充量等于或低于阈值且需要服务以防止欠载时，该中断被置位。通过软件可编程寄存器设置的阈值确定生成中断的发送 FIFO 条目级别。当数据写入发送 FIFO 缓冲区使其超过阈值时，该中断由硬件清除。

发送 FIFO 溢出中断 (ssi_txo_intr)

当 APB 访问尝试在发送 FIFO 已满后写入数据时触发。设置时，APB 写入的数据将被丢弃。该中断保持设置状态，直到读取发送 FIFO 溢出中断清除寄存器 (TXOICR) 为止。

接收 FIFO 满中断 (ssi_rxf_intr)

当接收 FIFO 达到阈值加 1 或以上时触发，需进行服务以防止溢出。通过软件可编程寄存器设置的阈值确定生成接收 FIFO 中断的条目级别。当从接收 FIFO 缓冲区读取数据使其低于阈值时，该中断由硬件清除。

接收FIFO溢出中断 (ssi_rxo_intr)

当接收逻辑尝试将数据放入已满的接收 FIFO 时，设置该中断。设置时，新接收的数据将被丢弃。该中断将保持设置状态，直到您读取接收 FIFO 溢出中断清除寄存器 (RXOICR)。

接收FIFO欠载中断 (ssi_rxu_intr)

当 APB 访问尝试从空的接收 FIFO 读取数据时，设置该中断。设置时，从接收 FIFO 读出的均为 0。该中断将保持设置状态，直到您读取接收 FIFO 欠载中断清除寄存器 (RXUICR)。

多主控争用中断 (ssi_mst_intr)

仅当 DW_apb_ssi 组件配置为串行主设备时，此中断才存在。当串行总线上的另一串行主设备将 DW_apb_ssi 主控设备选为串行从设备并主动传输数据时，设置该中断。此中断用于通知处理器串行总线可能发生争用。该中断将保持设置状态，直到您读取多主控中断清除寄存器 (MSTICR)。

组合中断请求 (ssi_intr)

屏蔽后所有上述中断请求的或运算结果。要屏蔽此中断信号，必须屏蔽所有其他 DW_apb_ssi 中断请求。

4.10.8. 传输模式

在串行总线上传输数据时，DW_apb_ssi 按本节所述模式运行。传输模式 (TMOD) 通过写入控制寄存器 0 (CTRLR0) 进行设置。

i 注意

传输模式设置不影响串行传输的双工模式。TMOD 对 Microwire 传输无效，Microwire 传输由 MWCR 寄存器控制。

4.10.8.1. 发送与接收

当 TMOD = **00b** 时，发送和接收逻辑均有效。数据传输依据所选帧格式（串行协议）正常进行。发送数据从发送 FIFO 弹出，经由 txd 线发送至目标设备，目标设备通过 rxd 线响应数据。目标设备的接收数据在每个数据帧结束时，从接收移位寄存器移入接收 FIFO。

4.10.8.2. 仅传输

当 TMOD = **01b** 时，接收数据无效，不应存储于接收 FIFO 中。数据传输将根据所选的帧格式（串行协议）正常进行。发送数据从发送 FIFO 弹出，经由 txd 线发送至目标设备，目标设备通过 rxd 线响应数据。在数据帧结束时，接收移位寄存器不会将其新接收的数据加载入接收 FIFO。接收移位寄存器中的数据

将被下一次传输覆盖。进入该模式时，应屏蔽来自接收逻辑的中断。

4.10.8.3. 仅接收

当 TMOD = **10b** 时，发送数据无效。作为从机配置时，发送 FIFO 在仅接收模式下不会弹出数据。传输过程中，txd 输出保持恒定逻辑电平。数据传输依据所选帧格式（串行协议）正常进行。目标设备的接收数据在每个数据帧结束时由接收移位寄存器移入接收 FIFO。进入该模式时，应屏蔽来自发送逻辑的中断。

4.10.8.4. EEPROM 读取

i 注意

此传输模式仅适用于主控配置。

当 TMOD = **11b** 时，传输的数据用于向 EEPROM 设备发送操作码和/或地址。通常这需要三个数据帧（8 位操作码，随后是 8 位高地址字节和 8 位低地址字节）。在发送操作码和地址期间，接收逻辑不会捕获任何数据（只要 DW_apb_ssi 主控通过其 txd 线发送数据，rxd 线上的数据即被忽略）。DW_apb_ssi 主控会持续发送数据，直到发送 FIFO 为空。因此，发送 FIFO 中应仅存放足够发送操作码和地址的数据帧；如果发送 FIFO 中的数据帧超过所需数量，读取数据将会丢失。

当发送 FIFO 变为空（所有控制信息已发送完毕）后，接收线（rxd）上的数据变为有效并存储至接收 FIFO；txd 输出保持在恒定的逻辑电平。串行传输将持续进行，直到 DW_apb_ssi 主控接收到的数据帧数量与 CTRLR1 寄存器中 NDF 字段的值加 1 相匹配。

i 注意

当 DW_apb_ssi 配置为 SSP 模式时，不支持 EEPROM 读取模式。

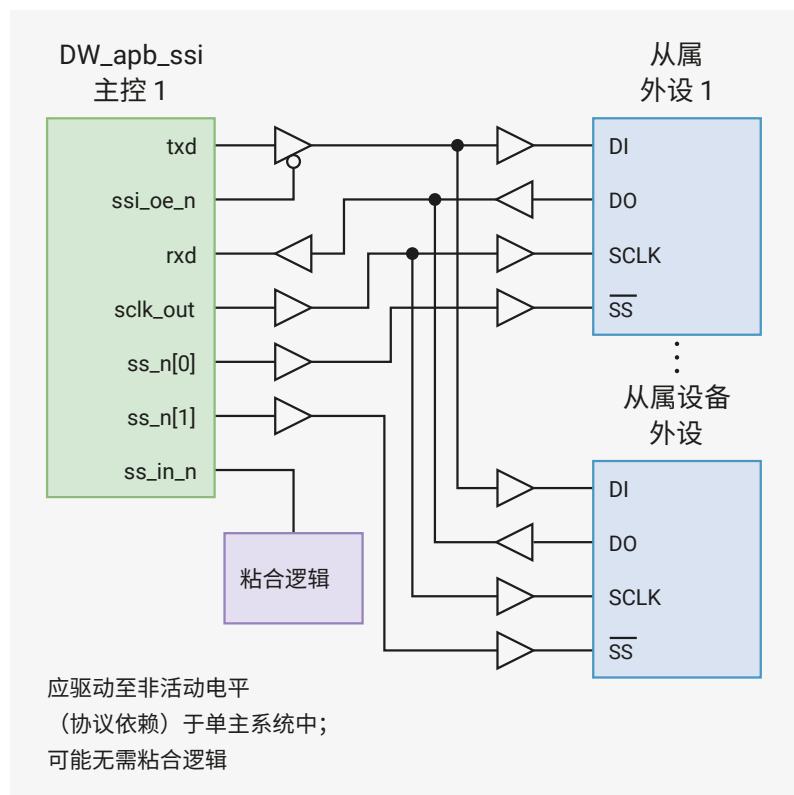
4.10.9. 操作模式

DW_apb_ssi 可配置为本节所述的基本操作模式。

4.10.9.1 串行主控模式

该模式支持与串行从属外设进行串行通信。当配置为串行主控设备时，DW_apb_ssi 负责发起并控制所有串行传输。图 12-0 展示了 DW_apb_ssi 配置为串行主控，且串行总线上的所有其他设备均配置为串行从属的示例。

图120.
DW_apb_ssi
*i*配置为主设备



串行比特率时钟由DW_apb_ssi生成和控制，并通过sclk_out线输出。当DW_apb_ssi禁用（SSI_EN = 0）时，不允许进行串行传输，且sclk_out保持在所用串行协议定义的“非活动”状态。

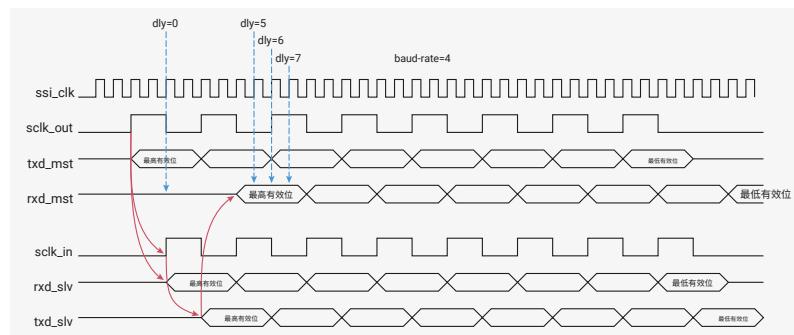
不支持多主配置。

4.10.9.1.1 RXD采样延迟

当DW_apb_ssi配置为主设备时，设计中可加入附加逻辑以延迟rxn信号的默认采样时间。该附加逻辑有助于提高串行总线的最大可实现频率。

主控设备的sclk_out信号与从属设备的rxn信号之间存在的往返路由延迟，可能导致主控设备看到的rxn信号时序偏离正常采样时间。图121示意了该情况。

图121。往返路由
延迟对sclk_out
信号的影响



从属设备使用主控设备的sclk_out信号作为触发信号，以驱动rxn信号数据传输至串行总线。

从属设备对sclk_out信号的路由及采样延迟，可能导致rxn位在主控设备采样该信号之前尚未稳定至正确值。图121展示了rxn信号路由延迟如何导致主控设备在默认采样时间采样到错误rxn值的示例。

若无 RXD 采样延迟逻辑，用户需提高传输波特率，以确保 rxd 信号的建立时间处于可接受范围内；这会导致串行接口频率降低。

当包含 RXD 采样延迟逻辑时，用户可以动态设置延迟值，以将 rxd 信号的采样时间从默认位置移动若干 ssi_clk 时钟周期。

采样延迟逻辑的分辨率为一个 ssi_clk 时钟周期。软件可通过编写循环程序，不断从从设备读取数据并递增主设备的 RXD 采样延迟值，从而“训练”串行总线，直到主设备正确接收数据。

4.10.9.1.2. 数据传输

数据传输由串行主设备发起。当 DW_apb_ssi 启用 (`SSI_EN=1`)，传输 FIFO 中至少存在一条有效数据，且已选择串行从设备时。在主动传输数据时，状态寄存器 (SR) 中的忙标志 (BUSY) 被置位。必须等待忙标志清除后，方可尝试新一轮串行传输。

注意

当数据写入发送FIFO时，BUSY状态不会被置位。仅当目标从设备被选中且传输正在进行时，该位才会被置位。在向发送FIFO写入数据后，移位逻辑不会开始串行传输，直到检测到sclk_out信号的上升沿。等待该上升沿的延迟取决于串行传输的波特率。在查询BUSY状态之前，应先查询TFE状态（等待其为1）或等待BAUDR * ssi_clk个时钟周期。

4.10.9.1.3. 主SPI和SSP串行传输

当传输模式为“发送并接收”或“仅发送”（分别对应`TMOD = 00b`或`TMOD = 01b`）时，移位控制逻辑将在发送FIFO为空时终止传输。对于连续数据传输，必须确保在所有数据传输完成之前，发送FIFO缓冲区不会变为空。传输FIFO阈值级别 (TXFT LR) 可用于提前中断处理器 (`ssi_txe_intr`)，以指示传输FIFO缓冲区接近空闲状态。当APB访问采用DMA时，传输数据级别 (DMATDLR) 可用于提前请求DMA控制器 (`dma_tx_req`)，以表明传输FIFO接近空闲状态。随后可以用数据重新填充FIFO，以继续串行传输。用户也可在启用串行从设备之前，向传输FIFO写入至少两个FIFO条目的数据块。这确保在传输FIFO中积累足够构成连续传输的数据帧数量之前，串行传输不会启动。

当传输模式设为“仅接收”（`TMOD = 10b`）时，选择串行从设备时通过向传输FIFO写入一个“虚拟”数据字以启动串行传输。`DW_apb_ssi`的txd输出在整个串行传输过程中保持恒定逻辑电平。发送FIFO仅在开始时弹出一次，且可能在整个串行传输过程中保持为空。串行传输的结束由控制寄存器1 (CTRLR1) 中的“数据帧数” (NDF) 字段决定。

例如，若您希望从串行从设备接收24个数据帧，应将NDF字段设为23；当接收的数据帧数等于NDF值加1时，接收逻辑即终止串行传输。此传输模式提升了APB总线的带宽，因为传输期间无需为发送FIFO提供服务。每当接收FIFO产生FIFO满中断请求时，必须及时读取接收FIFO缓冲区以防止溢出。

当传输模式为“eeprom_read”（`TMOD = 11b`）时，选定串行从设备（EEPROM）后，通过写入操作码和/或地址至发送FIFO启动串行传输。操作码和地址传输至EEPROM设备，随后从EEPROM设备接收数据并存储于接收FIFO中。串行传输的结束由控制寄存器1 (CTRLR1) 中的NDF字段控制。

注意

当 DW_apb_ssi 配置为 SSP 模式时，不支持 EEPROM 读取模式。

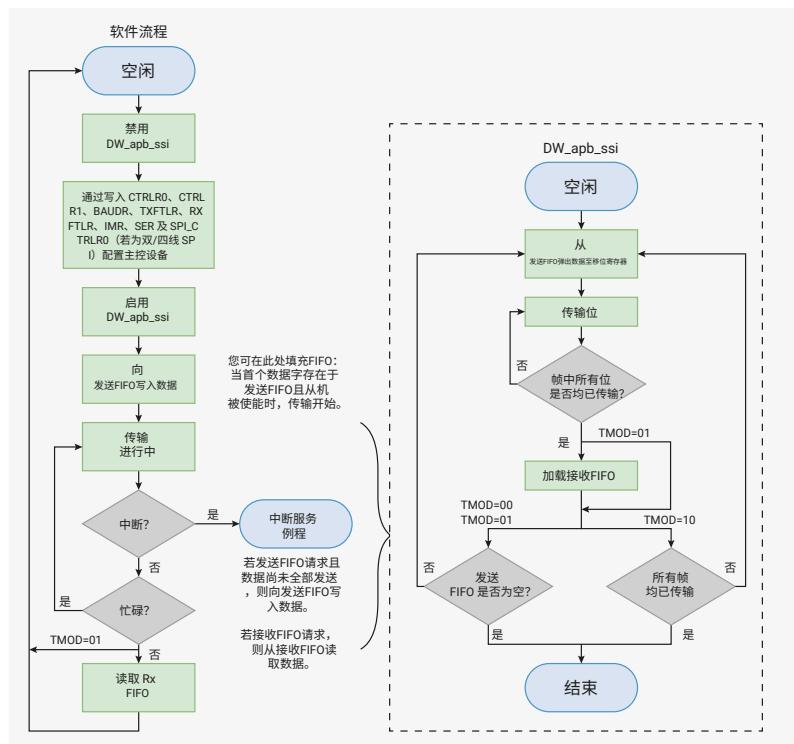
接收FIFO阈值水平（RXFTLR）可用于提前指示接收FIFO即将满载。当APB访问使用DMA时，接收数据水平（DMARDLR）可用于提前请求（dma_rx_req）DMA控制器，指示接收FIFO即将满载。

完成来自DW_apb_ssi串行主设备的SPI或SSP串行传输的典型软件流程如下所示：

1. 若DW_apb_ssi已启用，应通过向SSI使能寄存器（SSIENR）写入0以禁用它。
2. 配置DW_apb_ssi的传输控制寄存器；这些寄存器可按任意顺序设置。
 - 写入控制寄存器0（CTRLR0）。对于SPI传输，必须将串行时钟极性和串行时钟相位参数设置为与目标从机设备一致。
 - 如果传输模式为仅接收，则将CTRLR1（控制寄存器1）写为传输帧数减一；例如，若要接收四个数据帧，则在CTRLR1中写入“3”。
 - 写入波特率选择寄存器（BAUDR）以设置传输波特率。
 - 写入发送和接收FIFO阈值寄存器（分别为TXFTLR和RXFTLR）以设置FIFO阈值。
 - 写入IMR寄存器以设置中断屏蔽。
 - 此处可写入从机使能寄存器（SER）以启用目标从机设备的选通。若此处启用从机，则只要发送FIFO中存在一个有效数据项，传输即开始。若在写入数据寄存器（DR）之前未启用任何从机，传输将在启用从机后开始。
3. 通过向 SSIENR 寄存器写入 1 以启用 DW_apb_ssi。
4. 将要传输的数据写入发送 FIFO（写 DR）以发送至目标从设备。若此时 SER 寄存器中未启用任何从设备，请立即启用以开始传输。
5. 轮询 BUSY 状态以等待传输完成。BUSY 状态不可立即轮询。
6. 若发生发送 FIFO 空中断请求，写入发送 FIFO（写 DR）；若发生接收 FIFO 满中断请求，读取接收 FIFO（读 DR）。
7. 当发送 FIFO 为空时，移位控制逻辑会停止传输。若传输模式为仅接收（TMOD = **10b**），当接收指定数量的帧后，移位控制逻辑会停止传输。传输完成后，BUSY 状态被重置为 0。
8. 若传输模式非仅发送（TMOD != **01b**），则读取接收 FIFO 直至为空。
9. 通过向SSIENR寄存器写入0以禁用DW_apb_ssi。

图122展示了启动DW_apb_ssi主控SPI/SSP串行传输的典型软件流程。该图亦显示了串行主控组件内部的硬件流程。

图 122。
DW_apb_ssi 主控设备
SPI/SSP 传输流程



4.10.9.1.4. 主控 Microwire 串行传输

DW_apb_ssi 串行主控设备的 Microwire 串行传输由 Microwire 控制寄存器 (MWCR) 控制。

MWHS 位域用于启用或禁用 Microwire 握手接口。MDD 位域控制数据帧方向（控制帧始终由主控设备发送，由从属设备接收）。MWMOD 位域定义传输模式为顺序或非顺序。

当传输 FIFO 中至少有一个控制字且从设备被使能时，DW_apb_ssi 串行主设备将启动所有 Microwire 传输。当 DW_apb_ssi 主设备发送数据帧 (MDD = 1) 时，移位逻辑将在传输 FIFO 为空时终止传输。当 DW_apb_ssi 主设备接收数据帧 (MD D = 1) 时，传输的终止取决于 MWMOD 位字段的设置。如果传输为非连续传输 (MWMOD = 0)，则在从从机移入数据帧后且传输 FIFO 为空时终止传输。当传输为连续传输 (MWMOD = 1) 时，移位逻辑将在接收的数据帧数等于 CTRLRO 寄存器中的值加 1 时终止传输。

当 DW_apb_ssi 主设备上的握手接口被使能 (MWHS = 1) 时，传输完成后将轮询目标从机的状态。仅当从设备报告处于就绪状态时，DW_apb_ssi 主设备才完成传输并清除其 BUSY 状态。若传输为连续模式，则在从设备返回就绪状态之前，不发送下一控制或数据帧。

完成 DW_apb_ssi 串行主设备的 Microwire 串行传输的软件典型流程如下：

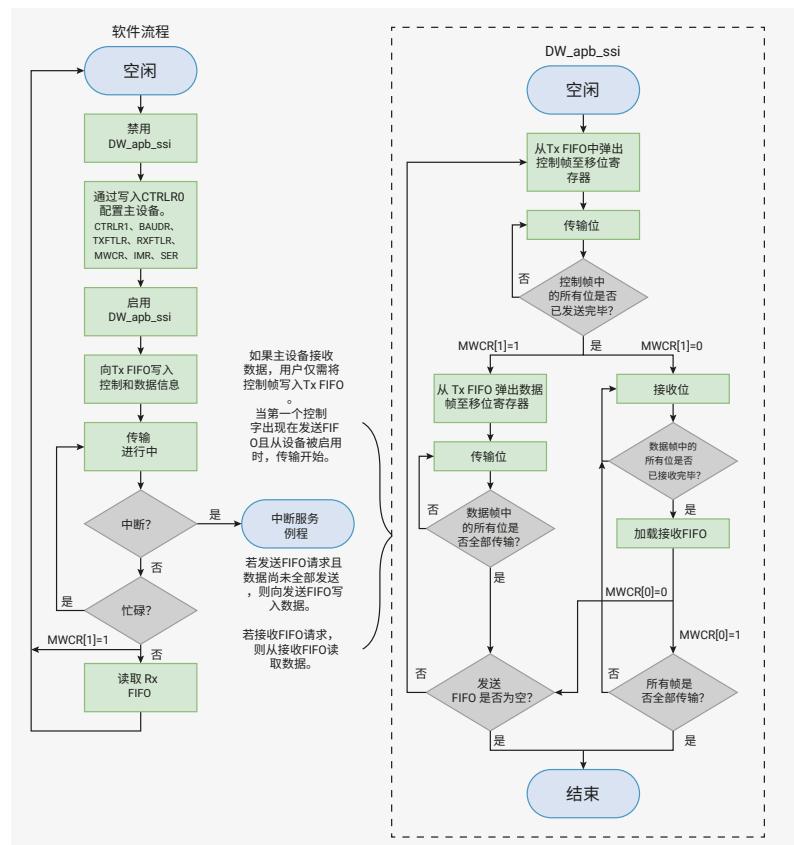
1. 若 DW_apb_ssi 已启用，则通过向 SSIENR 寄存器写入 0 来禁用该模块。
2. 配置 DW_apb_ssi 控制寄存器以准备传输。上述寄存器可以按任意顺序配置。向 CTRLRO 寄存器写入数据以设置传输参数。
 - 当传输为顺序模式且 DW_apb_ssi 主设备接收数据时，向 CTRLR1 寄存器写入传输帧数减一的值；例如，若要接收四帧数据，应向 CTRLR1 写入“3”。
 - 向 BAUDR 寄存器写入数据以设置传输波特率。
 - 向 TXFTLR 和 RXFTLR 寄存器写入数据以设置 FIFO 阈值电平。
 - 写入 IMR 寄存器以设置中断屏蔽。

您可以写入SER寄存器以使目标从设备被选中。如果此处启用了某个从设备，只要传输FIFO中有一个有效数据项，传输即开始。如果在写入DR寄存器之前未启用任何从设备，则传输将在启用从设备后才开始。

3. 通过向SSIENR寄存器写入1以启用DW_apb_ssi。
 4. 若DW_apb_ssi主设备发送数据，应将控制字和数据字写入传输FIFO（写入DR）。若DW_apb_ssi主设备接收数据，应将控制字写入传输FIFO。
- 若此时SER寄存器中未启用任何从设备，请立即启用以开始传输。
5. 轮询BUSY状态以等待传输完成。BUSY状态不可立即轮询。
 6. 当发送FIFO为空时，移位控制逻辑会停止传输。当传输模式为顺序且DW_apb_ssi主设备接收数据时，移位控制逻辑将在接收指定数量的数据帧后停止传输。传输完成后，BUSY状态被重置为0。
7. 若DW_apb_ssi主设备接收数据，需读取接收FIFO直至其为空。
 8. 通过向SSIENR寄存器写入0以禁用DW_apb_ssi。

图123展示了启动DW_apb_ssi主设备Microwire串行传输的典型软件流程。该图亦显示了串行主控组件内部的硬件流程。

图123。
DW_apb_ssi主控设备
Microwire传输
流



4.10.10. 对端连接接口

DW_apb_ssi可通过以下章节中讨论的任一接口连接至任何串行从属外设。

4.10.10.1. 摩托罗拉串行外设接口（SPI）

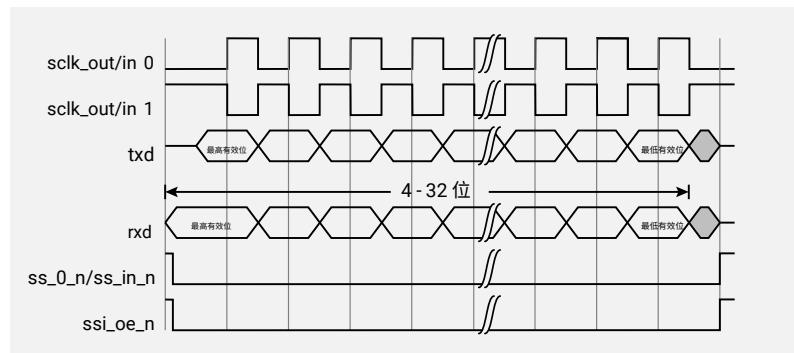
使用 SPI 时，时钟极性（SCPOL）配置参数决定串行时钟的非活动状态为高电平还是低电平。要传输数据，两个 SPI 外设必须具有相同的串行时钟相位（SCPH）和时钟极性（SCPOL）值。数据帧长度可为 4 至 16/32 位（具体取决于 SSI_MA_X_XFER_SIZE）。

当配置参数 SCPH=0 时，数据传输自从属选择信号的下降沿开始。

第一个数据位由主设备和从设备在串行时钟的第一个沿捕获；因此，有效数据须在第一个串行时钟沿到来之前存在于 txd 和 rxd 线路上。

图 124 显示了 SCPH=0 状态下单次 SPI 数据传输的时序图。序列时钟示意了配置参数 SCPOL = 0 及 SCPOL = 1 的情况。

图 124。SPI 串行格式 (SCPH = 0)



本节时序图中所示信号包括：

sclk_out

来自 DW_apb_ssi 主控的串行时钟

ss_0_n

来自 DW_apb_ssi 主控的从机选择信号

ss_in_n

输入至 DW_apb_ssi 从机的从机选择信号

ssi_oe_n

DW_apb_ssi 主控的输出使能信号

txd

DW_apb_ssi 主控的发送数据线

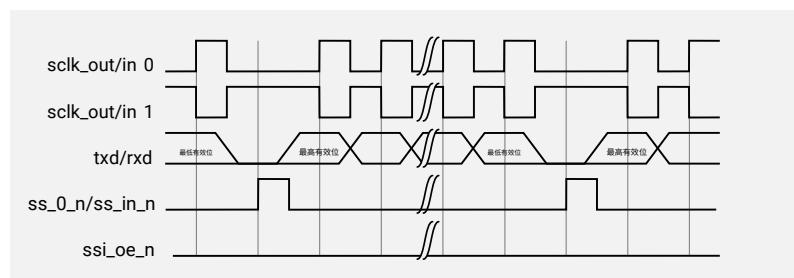
rxn

DW_apb_ssi 主控的接收数据线

当 SCPH = 0 时，支持连续数据传输：

- 当 CTRL0.SSTE 设为 1 时，DW_apb_ssi 会在帧间切换从机选择信号，且在从机选择信号有效期间，串行时钟保持默认值；该运行模式如图125所示。

图125。串行
格式连续
传输 (SCPH = 0)

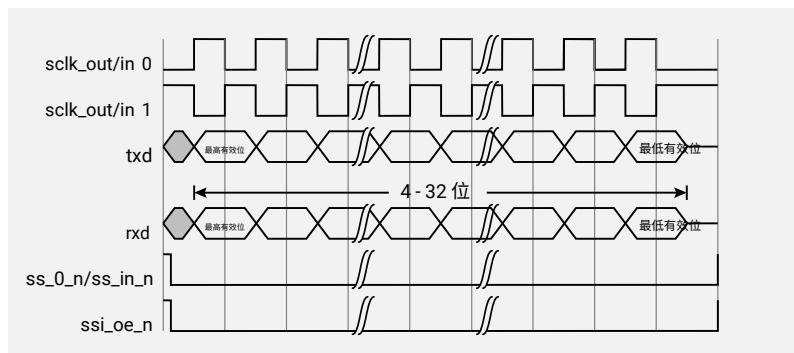


当配置参数 SCPH = 1 时，主设备外设在从选择线激活后的第一个串行时钟边沿开始传输数据。

从选择线激活后。第一个数据位在第二个（后沿）串行时钟边沿被捕获。数据由主设备外设在串行时钟的前沿传播。在连续数据帧传输期间，从选择线可保持低电平，直到最后一帧的最后一位被捕获。

图126展示了配置参数SCPH = 1时的SPI格式时序图。

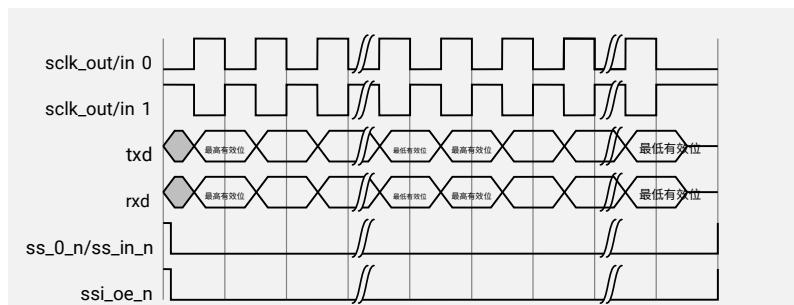
图126。 SPI串行
格式 (SCPH = 1)



连续数据帧的传输方式与单帧相同，下一帧的最高有效位紧接当前帧的最低有效位之后传输。从设备选择信号在整个传输过程中保持有效。

图127展示了配置参数SCPH = 1时连续SPI传输的时序图。

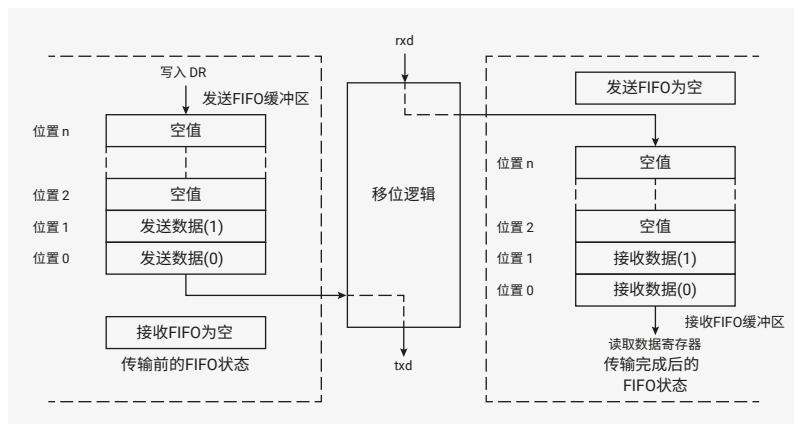
图127。 SPI串行
格式 连续
传输 (SCPH = 1)



DW_apb_ssi支持四种传输模式以执行SPI串行事务。对于发送和接收传输（控制寄存器0中传输模式字段(9:8) = 00b），从DW_apb_ssi发出的数据写入发送FIFO，从外部串行设备接收的数据存入接收FIFO。

图128展示了串行传输开始前及传输完成时的FIFO状态。此示例中，DW_apb_ssi通过连续传输方式向外部串行设备发送了两个数据字。外部串行设备亦会回复两个数据字，用于DW_apb_ssi。

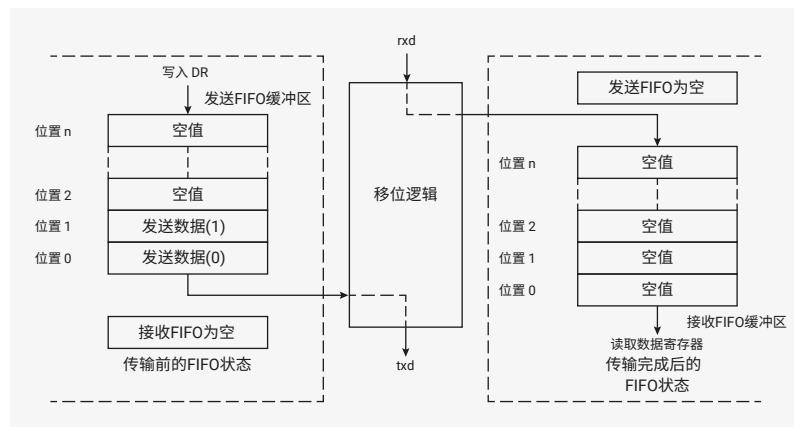
图128。 发送
与接收SPI及SSP传
输的FIFO状态



对于仅发送传输（控制寄存器0的传输模式字段 (9:8) = 01b），从DW_apb_ssi发送到外部串行设备的数据写入发送FIFO。由于从外部串行设备接收的数据被视为无效，故不存储于DW_apb_ssi的接收FIFO中。

图129展示了串行传输开始前及传输完成时的FIFO水位。在此示例中，两个数据字作为连续传输自DW_apb_ssi发送至外部串行设备。

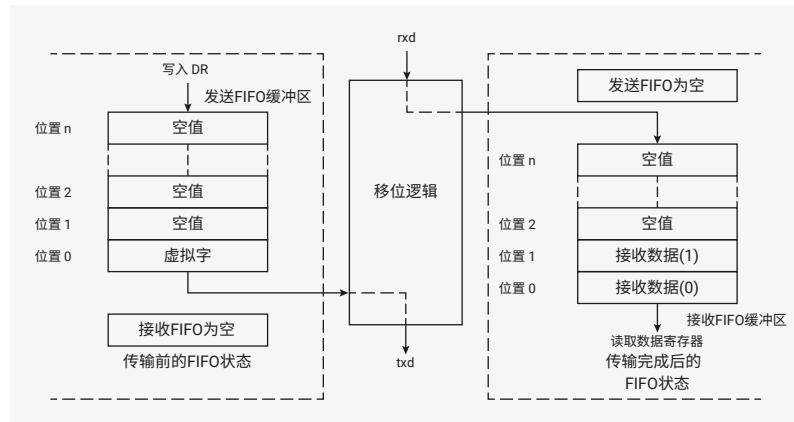
图129。仅针对发送的SPI和SSP传输的FIFO状态



对于仅接收传输（控制寄存器0的传输模式字段 $(9:8) = 10b$ ），从DW_apb_ssi发送至外部串行设备的数据无效，故写入单个伪字至发送FIFO以启动串行传输。DW_apb_ssi的txd输出在整个串行传输过程中保持恒定逻辑电平。来自外部串行设备的数据进入DW_apb_ssi后被推入接收FIFO。

图130展示了串行传输开始前以及传输完成时的 FIFO 水平。在此示例中，DW_apb_ssi 在连续传输过程中从外部串行设备接收了两个数据字。

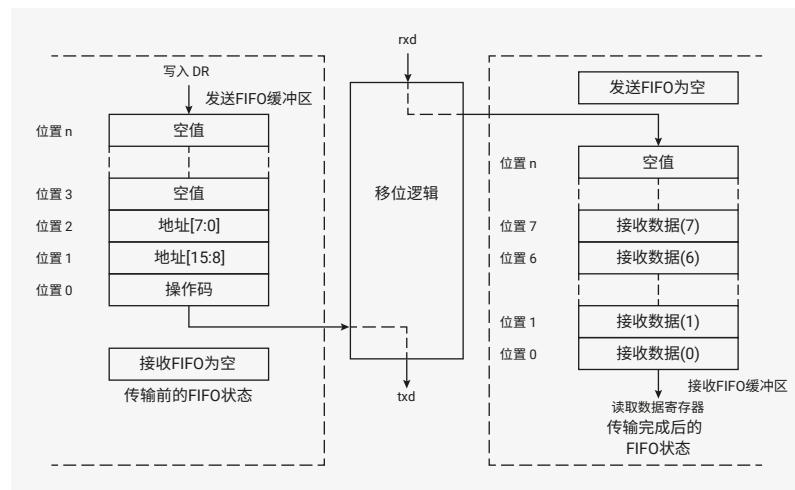
图130. FIFO接收状态
仅限 SPI 和 SSP 传输



对于 eeprom_read 传输（控制寄存器0中传输模式字段[9:8]为 11b），操作码和/或 EEPROM 地址写入发送 FIFO。在传输这些控制帧期间，DW_apb_ssi 主控不会捕获接收的数据。控制帧传输完成后，来自 EEPROM 的接收数据存储于接收 FIFO 中。

图131展示了串行传输开始前以及传输完成时的 FIFO 水平。在本例中，发送一个操作码及高位和低位地址到EEPROM，同时从EEPROM读取八个数据帧，并存储于DW_apb_ssi主控的接收FIFO中。

图131。EEPROM
M读取传输模式
下的FIFO状态

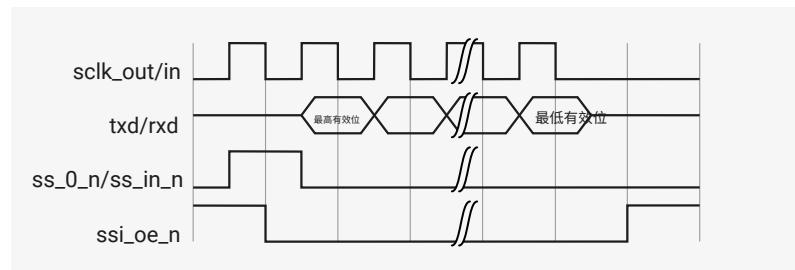


4.10.10.2. 德州仪器同步串行协议 (SSP)

数据传输始于将帧指示线 (ss_0_n/ss_in_n) 断言一个串行时钟周期。待传输的数据在一个串行时钟周期后驱动至txd线；从设备的数据同样驱动至rxd线。数据于串行时钟 (sclk_out/sclk_in) 上升沿传播，并于下降沿采样。数据帧长度范围为4至32位。

图132展示了单个SSP串行传输的时序图。

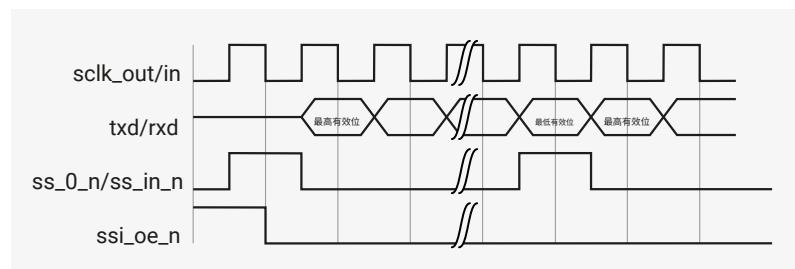
图132. SSP串行
格式



连续数据帧的传输方式与单个数据帧完全相同。帧指示信号在当前传输LSB的同一时钟周期内被确认一个时钟周期，表示后续还有数据帧。

图133展示了连续SSP传输的时序。

图133。SSP串行
格式连续
传输



4.10.10.3. 国家半导体Microwire协议

数据传输从从机选择信号 (ss_0_n) 下降沿开始。半个串行时钟周期 (sclk_out) 后，控制字的第一位通过txd线发送。控制字长度范围为1至16位，通过写入CTRLR0寄存器中位域CFS（第15至12位）进行设置。其余控制字由DW_apb_ssi串行主机在sclk_out下降沿传输。在此传输期间，串行主机的rxd线处于无数据（高阻抗）状态。

数据字的方向由Microwire控制寄存器（MWCR）中的MDD位字段（第1位）控制。当MDD=0时，表示DW_apb_ssi串行主机从外部串行从机接收数据。控制字最低有效位传输完成后一个时钟周期，从机外设响应一个虚拟0位，随后是长度为4至32位的数据帧。数据在串行时钟的下降沿传输，并于上升沿采样。

在传输过程中，从机选择信号保持低电平有效，数据传输完成后半个时钟周期解除使能。图134显示了单个DW_apb_ssi串行主机从外部串行从机读取的时序图。

图134。单通道
DW_apb_ssi主控
Microwire串行
传输（MDD=0）

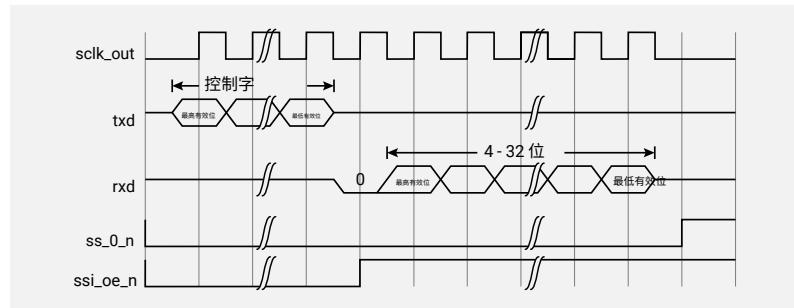
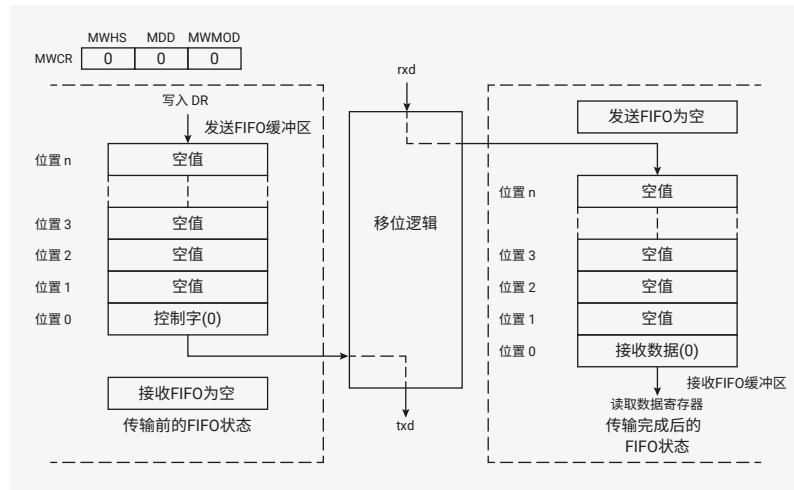


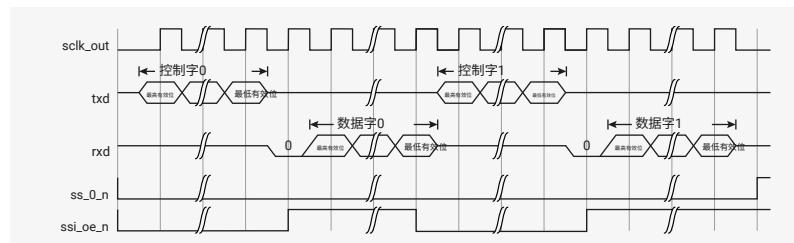
图135。单通道
Microwire传
输的FIFO状态（
接收数据帧）



Microwire协议的连续传输可以是顺序或非顺序的，由MWCR寄存器内的MWMOD位（第0位）控制。

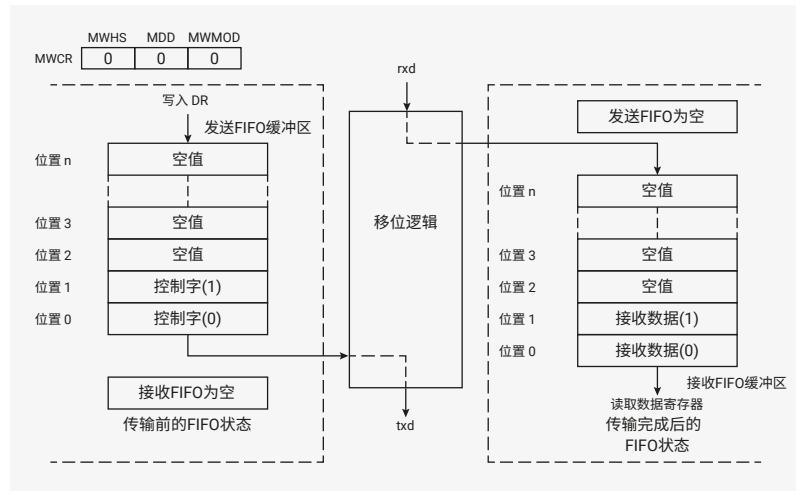
非顺序连续传输如图136所示，下一次传输的控制字紧贴当前数据字的最低有效位之后。

图136。
连续
非连续
Microwire传
输
(接收数据帧)



执行连续非连续传输所需的唯一修改是向发送 FIFO 缓冲区写入更多控制字；这在图 137 中有所体现。在本例中，从外部串行从设备读取了两个数据字。

图 137。非连续 Microwire 传输（接收数据帧）时的 FIFO 状态



在顺序连续传输过程中，仅由 DW_apb_ssi 主控发送一个控制字。传输的启动方式与非连续读取操作相同，但该周期将继续以读取更多数据。

从设备自动将地址指针递增至下一个位置，并继续从该位置提供数据。可通过此方式读取任意数量的位置；当接收的数据字数达到 CTRLR1 寄存器中数值加一时，DW_apb_ssi 主控终止传输。

图 138 的时序图及图 139 的示例展示了从外部从设备连续顺序读取两个数据帧的过程。

图138。
连续顺序
Microwire传输
(接收数据帧)

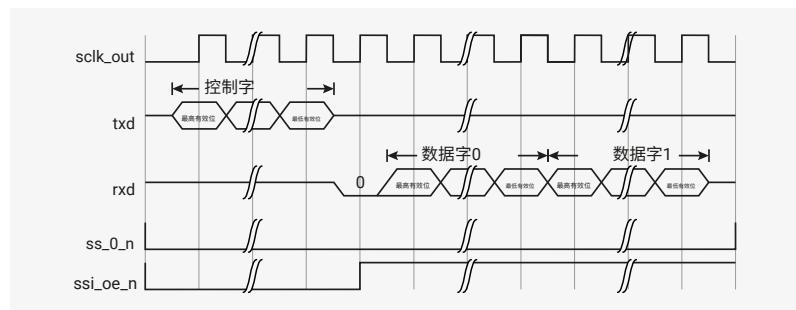
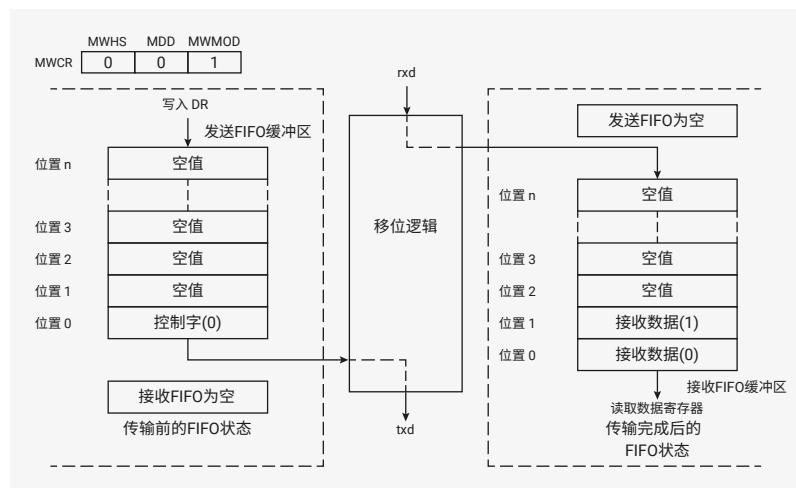


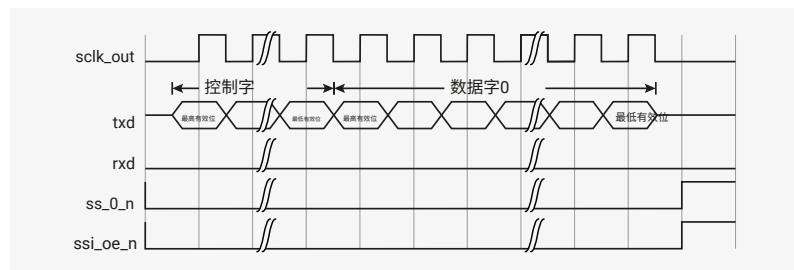
图139。顺序
Microwire传输的FIFO状态 (接收数
据帧)



当MDD=1时，表示DW_apb_ssi串行主设备向外部串行从设备发送数据。在控制字最低有效位传输完成后，DW_apb_ssi 主设备开始向从属外设传输数据帧。

图140显示了单个DW_apb_ssi串行主设备写入外部串行从设备的时序图。

图140。单次Microwire传输（发送数据帧）

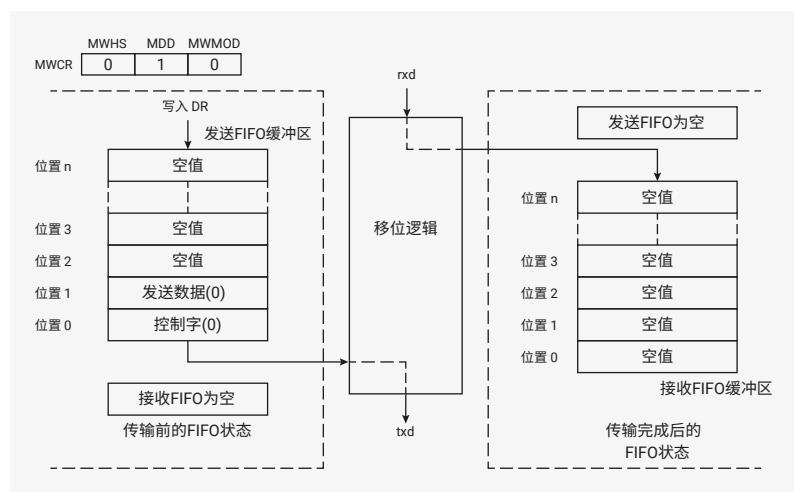


注意

DW_apb_ssi不支持连续顺序Microwire写操作，即MDD=1且MWMOD=1。

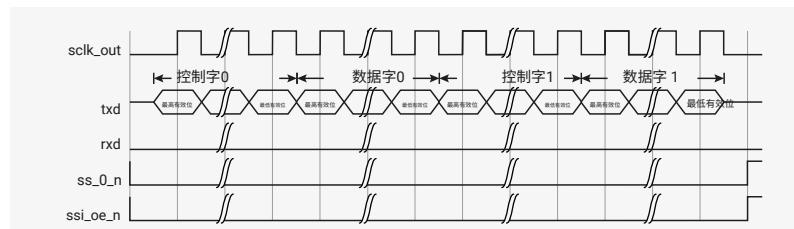
图141展示了传输前发送FIFO中数据帧和控制帧的结构，同时显示了MWCR寄存器中编程的值。

图141。单次Microwire传输的FIFO状态（传输数据帧）



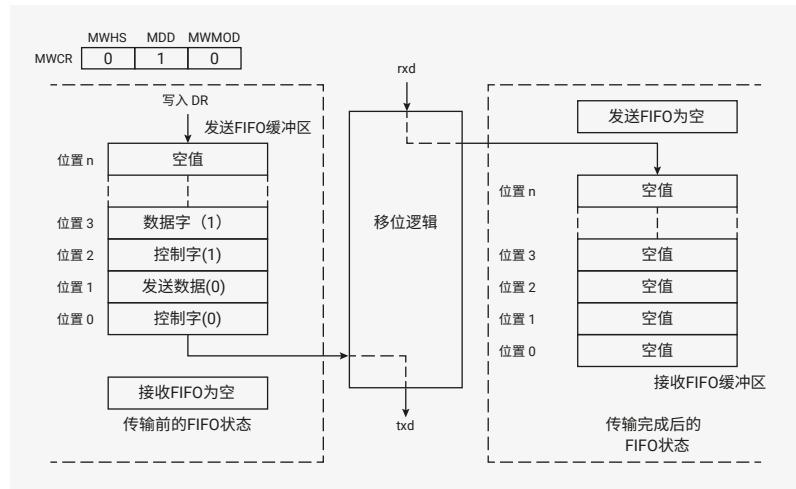
连续传输如图142所示，下一次传输的控制字紧跟当前数据字的最低有效位之后。

图142。
连续 Microwire 传输
(传输数据帧)



执行连续传输的唯一修改是向发送 FIFO 缓冲区写入更多控制字和数据字，见图143。该示例显示向外部串行从设备写入了两个数据字。

图143。连续
Microwire传输的FIFO状态（数据帧发送）

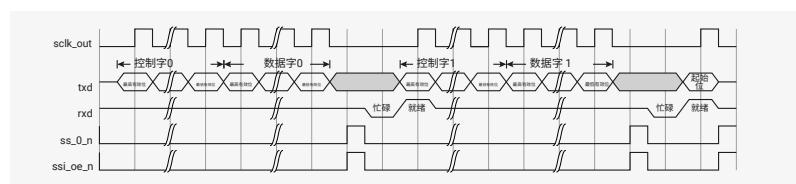


Microwire 握手接口也可用于 DW_apb_ssi 主设备对外部串行从设备的写操作。要启用握手接口，须在 MWCR 寄存器的 MHS 位域（第2位）写入1。

当 MHS 设置为 1 时，DW_apb_ssi 串行主控在完成传输前，或执行连续传输时发送下一个控制字之前，会先检查从设备的就绪状态。

图 144 展示了启用握手接口的连续 Microwire 传输示例。

图 144。
带握手的连续 Micro
wire 传输（
数据帧发送
）



第一个数据字发送至串行从设备后，DW_apb_ssi 主控轮询 rxd 输入端口，等待从设备返回就绪状态。接收到就绪状态后，DW_apb_ssi 主控开始发送下一个控制字。最后一个数据帧发送完成后，DW_apb_ssi 主控发送起始位以清除从设备就绪状态，然后完成传输。该传输的 FIFO 状态与图 143 相同，仅 MWHS 位字段被设置为 1。

要从 DW_apb_ssi 主设备向串行从设备传输控制字（且不跟随数据），传输 FIFO 缓冲区中必须仅包含一条数据。无法在连续传输中发送两个控制字，因为 DW_apb_ssi 的移位逻辑会将第二个控制字识别为数据字。当 DW_apb_ssi 主设备仅传输控制字时，须将 MWCR 寄存器第 1 位的 MDD 位设置为 1。

如图 145 所示示例及图 146 的时序图中，握手接口已启用。若握手接口禁用（MHS=0），则在从设备捕获控制字最低有效位之后的一个 sclk_out 周期内，DW_apb_ssi 主设备将终止传输。

图 145。Micro
wire控制字传输的
FIFO状态

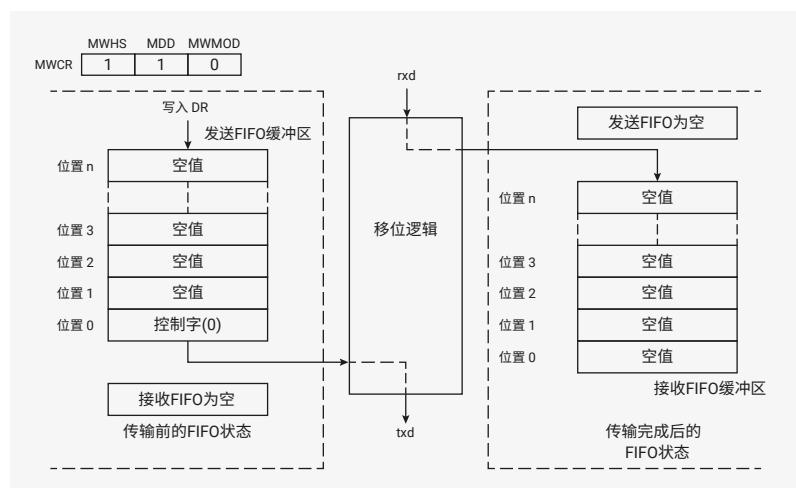
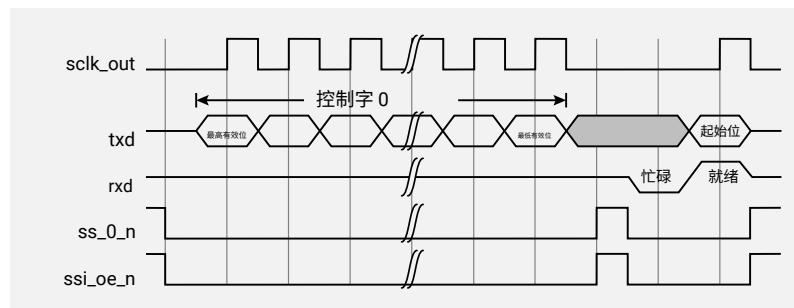


图146. Microwire
控制字



4.10.10.4. 增强型 SPI 模式

DW_apb_ssi 支持 RP2040 中 SPI 的双模和四模；不支持八模。txd、rxd 及 ssi_oe_n 信号宽度均为四位。

数据通过多条线路进行移出与移入，从而提升整体吞吐量。该模式支持串行时钟极性和相位的全部四种组合，且其行为与普通 SPI 模式相同。双 SPI 与四 SPI 模式功能相似，唯一差异在于 txd、rxd 及 ssi_oe_n 信号的宽度。操作模式（写入/读取）可通过 CTRLR0.TMOD 字段进行选择。

4.10.10.4.1. 增强型 SPI 模式中的写操作

双 SPI 或四 SPI 写操作可分为三个阶段：

- 指令阶段
- 地址阶段
- 数据阶段

以下寄存器字段用于写操作：

- CTRLR0.SPI_FRF — 指定帧传输的格式。
- SPI_CTRLR0 (控制寄存器0) — 指定指令、地址和数据的长度。
- SPI_CTRLR0.INST_L — 指定指令长度（可选值为0、4、8或16位）。
- SPI_CTRLR0.ADDR_L — 指定地址长度（详见表579的解释值）。
- CTRLR0.DFS 或 CTRLR0.DFS_32 — 指定数据长度。

一条指令占用一个FIFO位置。一个地址可能占用多个FIFO位置。

指令和地址必须均编入数据寄存器（DR）。DW_apb_ssi将等待指令和地址均编入后，才开始写操作。

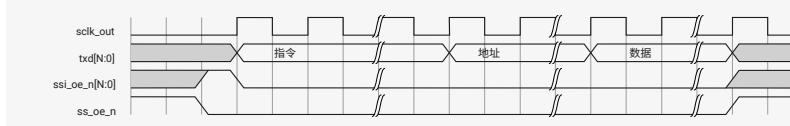
指令、地址及数据可在双线/四线模式下发送，该模式由SPI_CTRLR0.TRANS_TYPE和CTRLR0.SPI_FRF字段选择。

注意

- 若CTRLR0.SPI_FRF选择为“标准SPI格式”，则所有数据均以标准SPI模式发送，SPI_CTRLR0.TRANS_TYPE字段将被忽略。
- CTRLR0.SPI_FRF仅在CTRLR0.FRF被编程为`00b`时适用。

图147展示了在双线或四线SPI模式下的典型写操作。当SSI_SPI_MODE设置为3时，N值为7；设置为2时，N值为3；设置为1时，N值为1。对于一次写操作，指令和地址仅发送一次，随后发送DR中编程的数据帧，直到发送FIFO为空。

图147。典型写操作
双线/四线SPI模式

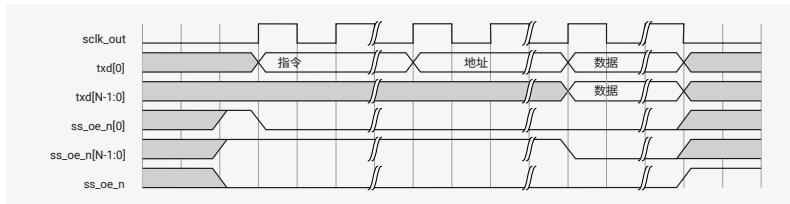


要启动双线/四线写操作，CTRLR0.SPI_FRF须分别设置为01、10或11。此操作将设置传输类型，且每条写命令的数据均按照CTRLR0.SPI_FRF字段指定的格式进行传输。

情况A：指令和地址均以标准SPI格式传输

为此，SPI_CTRLR0.TRANS_TYPE字段须设为`00b`。图148显示了指令和地址均以标准SPI格式传输时的时序图。当CTRLR0.SPI_FRF设为11b时，N值为7；设为10b时，N值为3；设为01b时，N值为1。

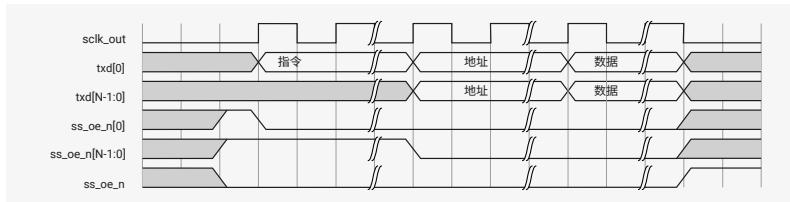
图148。指令和地址以标准SPI格式传输



情况B：指令以标准格式传输，地址以增强SPI格式传输

为此，必须将SPI_CTRLR0.TRANS_TYPE字段设置为1。图149展示了在指令以标准格式传输且地址以增强SPI格式传输时的时序图。当CTRLR0.SPI_FRF设置为`11b`时，N的值为7；设置为`10b`时，N的值为3；设置为`01b`时，N的值为1。

图149。指令以标准格式传输，地址以增强SPI格式传输

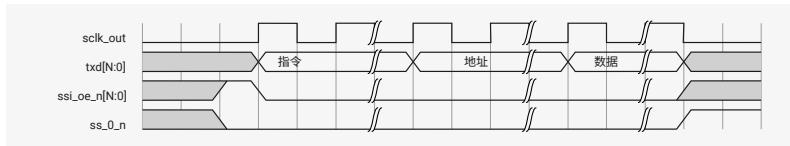


情况C：指令和地址均以增强SPI格式传输

为此，必须将SPI_CTRLR0.TRANS_TYPE字段设置为`10b`。图150展示了指令和地址以CTRLR0.SPI_FRF字段指定之SPI格式传输的时序图。N的值为：

当CTRLR0.SPI_FRF设置为`11b`时，N的值为7；设置为`10b`时为3；设置为`01b`时为1。

图150。增强SPI格式中同时传输指令和地址

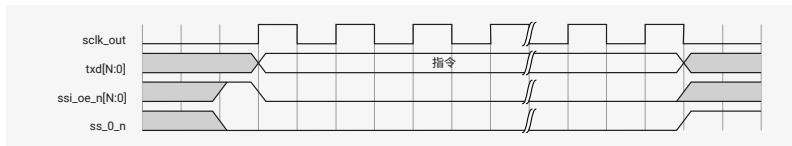


情况D：增强SPI格式中仅传输指令

为此，SPI_CTRLR0.TRANS_TYPE字段必须设置为`10b`。图151显示了该传输的时序图。N的值为：当CTRLR0.SPI_FRF设置为`11b`时为7；设置为`10b`时为3；且当

CTRLR0.SPI_FRF 设置为 **01b** 时为 1。

图 151。增强 SPI 格式中仅传输指令



4.10.10.4.2. 增强 SPI 模式下的读取操作 双线或四线 SPI

读取操作可分为四个阶段：

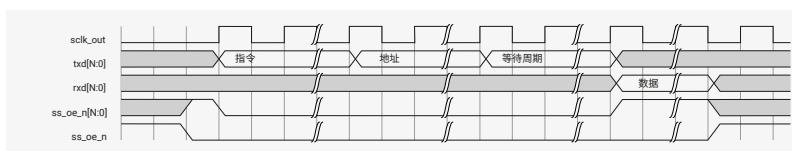
- 指令阶段
- 地址阶段
- 等待周期
- 数据阶段

等待周期可通过 SPI_CTRLR0.WAIT_CYCLES 字段进行编程。编程进 SPI_CTRLR0.WAIT_CYCLES 的值直接对应于 sclk_out 的周期数。例如，WAIT_CYCLES=0 表示无等待，WAIT_CYCLES=1 表示等待一个周期，依此类推。等待周期用于使目标从机从输入模式切换至输出模式，不同设备的等待周期可能不同。

对于读取操作，DW_apb_ssi 一次性发送指令和控制数据，等待直到接收 NDF（CTRLR1 寄存器）指定数量的数据帧，随后取消从机选择信号。

图152展示了双四线 SPI 模式下的典型读取操作。当 SSI_SPI_MODE 设为四线模式时，N 的值为3；当 SSI_SPI_MODE 设为双线模式时，N 的值为1。

图152。增强SPI模式下的典型读取操作



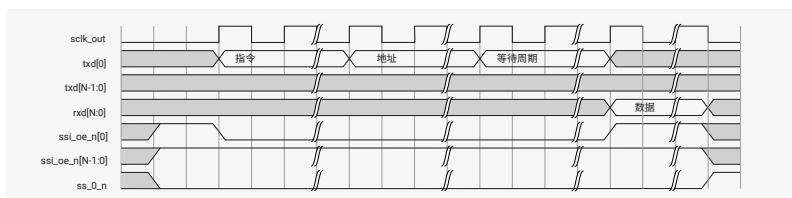
要启动双路/四路读取操作，CTRLR0.SPI_FRF 必须分别设置为 01、10 或 11。此设置确定传输类型，之后每个读取命令的数据将按照 CTRL0.SPI_FRF 字段指定的格式传输。

以下为增强SPI模式下写操作的可能情况：

情况A：指令和地址均以标准SPI格式传输

为此，SPI_CTRLR0.TRANS_TYPE 字段应设置为 **00b**。图153显示了指令和地址均以标准SPI格式传输时的时序图。图中还展示了地址之后的等待周期，可通过 SPI_CTRLR0.WAIT_CYCLES 字段进行编程。当 CTRLR0.SPI_FRF 设置为 **11b** 时，N 的值为 7；设置为 **10b** 时，N 为 3；设置为 **01b** 时，N 为 1。

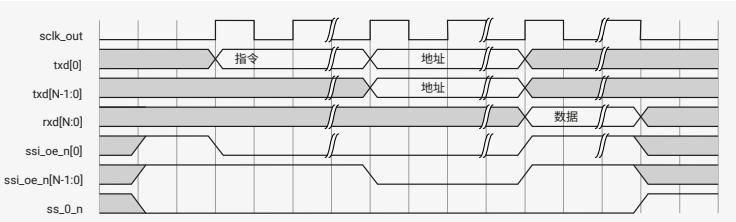
图153。指令和地址以标准SPI格式传输



情况B：指令以标准格式传输，地址以双路SPI格式传输

为此，应将 SPI_CTRLR0.TRANS_TYPE 字段设置为 **01b**。图154显示了指令以标准格式传输，地址以双路SPI格式传输的时序图。当 CTRLR0.SPI_FRF 设置为 **11b** 时，N 的值为 7；设置为 **10b** 时，N 为 3；设置为 **01b** 时，N 为 1。

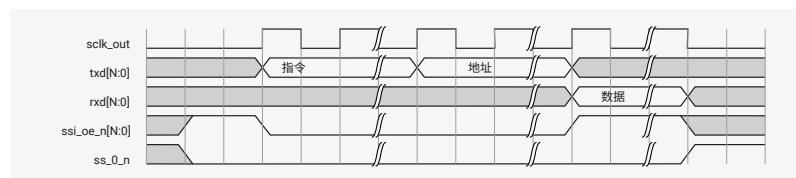
图154。指令以标准格式传输，地址以增强 SPI 格式传输



情况C：指令和地址均以双路SPI格式传输

为此，`SPI_CTRLR0.TRANS_TYPE` 字段必须设置为 `10b`。图155显示了指令和地址均以双路SPI格式传输的时序图。当 `CTRLR0.SPI_FRF` 设置为 `11b` 时，N 的值为 7；设置为 `10b` 时，N 的值为 3；设置为 `01b` 时，N 的值为 1。

图155。指令及地址以增强型SPI格式传输



情况D：无指令、无地址的读操作传输

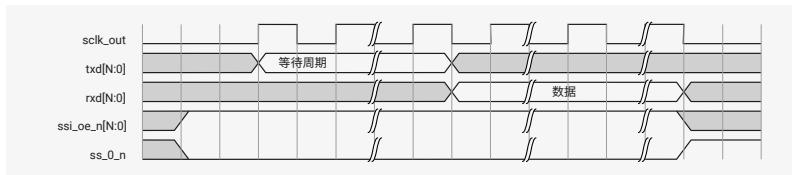
为此，`SPI_CTRLR0.ADDR_L`与`SPI_CTRLR0.INST_L`须设置为0，且`SPI_CTRLR0.WAIT_CYCLES`须设置为非零值。表5-79列出了`ADDR_L`解码值及其对应的增强（双/四）SPI模式说明。

表579. `ADDR_L`增强模式下的解码
SPI模式

<code>ADDR_L</code> 解码值	描述
0000	0位地址宽度
0001	4位地址宽度
0010	8位地址宽度
0011	12位地址宽度
0100	16位地址宽度
0101	20位地址宽度
0110	24位地址宽度
0111	28位地址宽度
1000	32位地址宽度
1001	36位地址宽度
1010	40位地址宽度
1011	44位地址宽度
1100	48位地址宽度
1101	52位地址宽度
1110	56位地址宽度
1111	60位地址宽度

图156显示了该类型传输的时序图。若`CTRLR0.SPI_FRF`设置为 `11b`，则N值为7；若设置为 `10b`，则N值为3；若设置为 `01b`，则N值为1。要启动此传输，软件必须在数据寄存器（DR）中执行虚拟写操作，`DW_apb_ssi`将等待设定的等待周期，然后读取NDF字段指定的数据量。

图156。无
指令且无
地址读
传输



4.10.10.4.3. 增强SPI模式的高级I/O映射

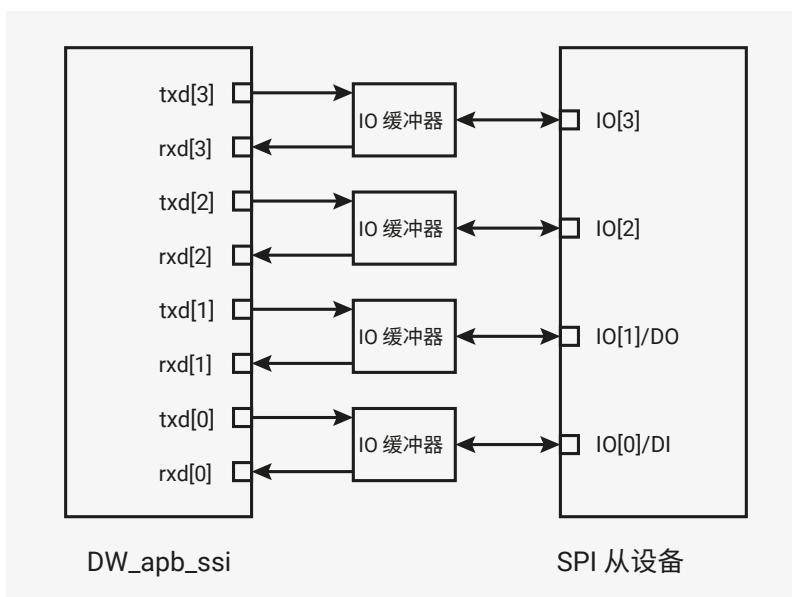
增强SPI模式（双线和四线）的输入输出映射已硬编码于DW_apb_ssi中。在标准SPI工作模式下，rx[1]信号用于采样接收数据。

对于其他协议（如 SSP 和 Microwire），I/O 映射保持不变。因此，其他协议能够轻松连接任何支持双通道/四通道 SPI 操作的设备，因为这些协议不需要设计外部存在多路复用（MUX）逻辑。

[图157展示了 DW_apb_ssi 在四通道模式下与另一支持四通道模式的 SPI 设备的 I/O 映射关系。](#)

如图157所示，IO[1] 引脚在标准 SPI 操作模式中用作 DO，且连接至 rx[1] 引脚，该引脚将在标准操作模式中采样输入信号。

图157. 高级
四路中的I/O映射
SPI模式



4.10.10.5. SPI操作中的双数据率（DDR）支持

在标准操作中，SPI模式下的数据传输发生于时钟的上升沿或下降沿。为提高吞吐量，可使用双数据率传输进行存储器读写。

DDR模式支持以下SPI协议模式：

- SCPH=0 且 SCPOL=0（模式0）
- SCPH=1 且 SCPOL=1（模式3）

DDR命令允许数据在时钟的双边沿传输。以下为不同类型的DDR命令：

- 地址和数据以DDR格式传输（数据则为接收），而指令以标准格式传输。
- 指令、地址与数据均以DDR格式传输或接收。

DDR_EN (SPI_CTRLR0[16]) 位用于确定地址和数据是否以DDR模式传输，INST_DDR_EN (SPI_CTRLR0[17]) 位用于确定指令是否必须以DDR格式传输。上述位

仅当 CTRLR0.SPI_FRF 位设置为双模式或四模式时有效。

图158描述了一种DDR写传输，其中指令持续以标准格式传输。在图158中，当 CTRLR0.SPI_FRF 设置为 11b时，N 的值为 7；设置为 10b时，N 的值为3；设置为 01b时，N 的值为1。

图158。 DDR
传输， SCPOL=0
且 SCPOL=0

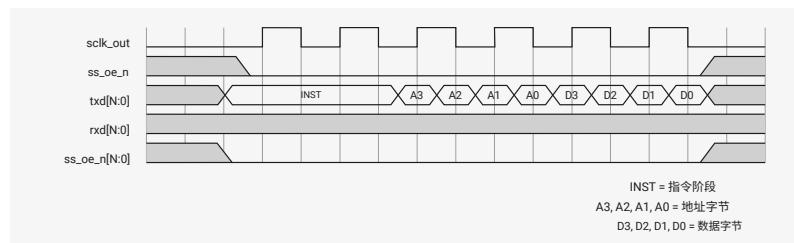
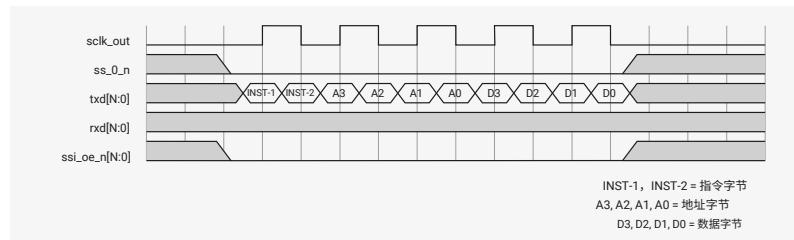


图159描述了一种DDR写传输，其中指令、地址和数据均以DDR格式传输。

图159。 指令
、地址及数
据以DDR格式传输



注意

在DDR传输中，地址和指令不能设置为0值。

4.10.10.5.1. DDR模式下的数据传输

在DDR模式下，数据在两个时钟边沿传输，因此难以准确采样数据。DW_apb_ssi使用内部寄存器确定数据应传输的时钟边沿。此设置确保接收端在采样时能获得稳定的数据。内部寄存器（DDR_DRIVE_EDGE）决定数据传输的时钟边沿。DW_apb_ssi根据波特率时钟发送数据，波特率时钟是内部时钟（ssi_clk * BAUDR）的整数倍。数据需在半个时钟周期（BAUDR/2）内传输，故DDR_DRIVE_EDGE的最大值为[(BAUDR/2)-1]。若 DDR_DRIVE_EDGE 编程值为 0，则数据相对于 sclk_out（波特率时钟）边沿进行传输对齐。若 DDR_DRIVE_EDGE 编程值为 1，则数据将在 sclk_out 边沿前一个 ssi_clk 时钟周期传输。

注意

若波特率被设定为 2，则数据始终边沿对齐。

图 160、图 161 及图 162 展示了使用不同 DDR_DRIVE_EDGE 寄存器值进行数据传输的示例。这些示例中的绿色箭头头标示数据驱动点。所有示例中使用的波特率均为 12。在图 160 中，数据传输边沿与驱动边沿一致。此为 DDR 模式下的默认行为。

图 160。 DDR_DRIVE
EDGE =
0时发送数据

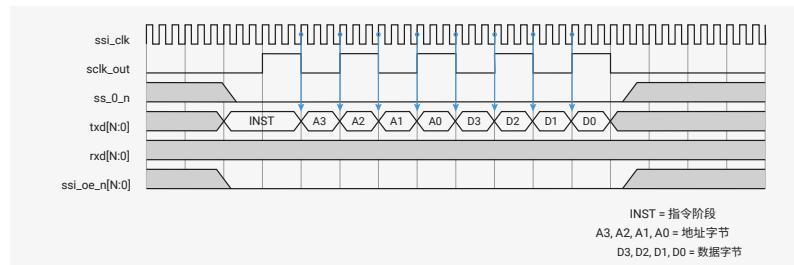


图 160 显示了传输边沿与驱动边沿一致的默认行为。

图 161。DDR_DRIV

E_EDGE =
1时发送数据

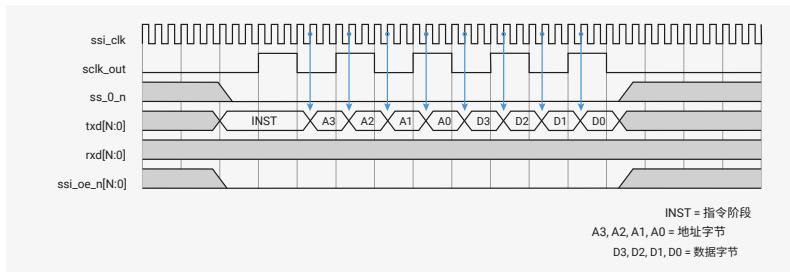
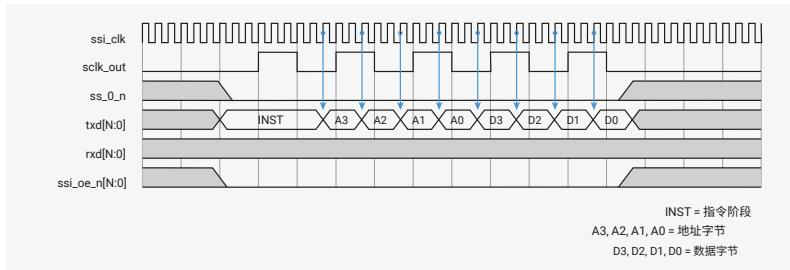


图 162。DDR_DRIV

E_EDGE =
2时发送数据



4.10.10.6. SPI 模式下的 XIP 模式支持

就地执行 (eXecute In Place, XIP) 模式允许通过 APB 接口直接传输 SPI 数据，而无需写入 DW_apb_ssi 的数据寄存器。当启用 XIP 缓存时，DW_apb_ssi 中将启用 XIP 模式。该控制信号指示 APB 传输是寄存器读写还是 XIP 读取操作。处于 XIP 模式时，DW_apb_ssi 仅在 APB 接口接收读请求。该请求将被转换为串行接口上的 SPI 读，数据接收后立即以同一事务返回 APB 接口。

① 注意

- XIP 操作期间仅支持 APB 读取。

地址长度由 SPI_CTRLR0.ADDR_L 字段确定，paddr 的相关位 ([SPI_CTRLR0.ADDR_L-1:0]) 作为地址传输至 SPI 接口。XIP 地址由 XIP 缓存控制器管理。

4.10.10.6.1 XIP 模式下的读操作

XIP 操作仅支持增强型 SPI 模式（双线和四线模式）下的操作。因此，CTRLR0.SPI_FRF 位不得被设置为 0。XIP 读操作分为两个阶段：

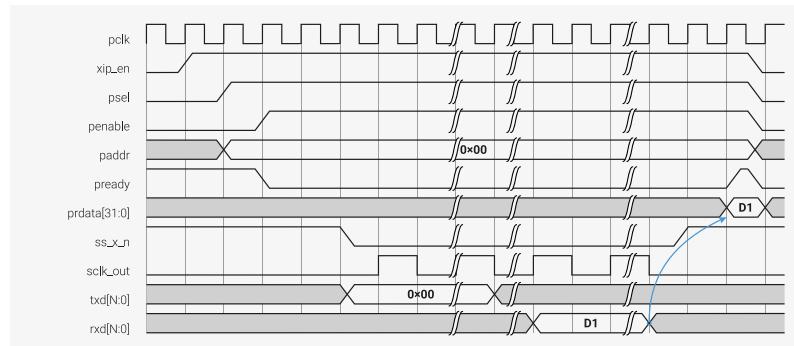
- 地址阶段
- 数据阶段

针对 XIP 读操作

1. 在 CTRLR0 寄存器中设置 SPI 帧格式和数据帧大小。注意，最大数据帧大小为 32。
2. 在 SPI_CTRLR0 寄存器中设置地址长度、等待周期及事务类型。注意，最大地址长度为 32。

完成上述设置后，用户可通过 APB 接口发起读事务，该事务将基于预设值传输至 SPI 外设。图 163 显示典型的 XIP 传输过程。N 的取值分别为：双线 SPI 模式为 1，四线 SPI 模式为 3 和 7。

图163。典型示例
XIP中的读取操作
模式



4.10.11. DMA 控制器接口

DW_apb_ssi具备内置的DMA功能；其具备与DMA控制器通信的握手接口，用以请求和控制数据传输。APB总线用于执行与DMA之间的数据传输。

i 注意

当DW_apb_ssi与DMA控制器连接时，DMA控制器始终作为流量控制器；即负责控制块的大小。该功能必须由DMA控制器中的软件编程实现。

DW_apb_ssi使用两个DMA通道，分别用于发送和接收数据。DW_apb_ssi包含以下DMA寄存器：

DMACR

用于启用DMA操作的控制寄存器。

DMATDLR

用于设置发送FIFO达到该级别时发出DMA请求的寄存器。

DMARDLR

用于设置接收FIFO达到该级别时发出DMA请求的寄存器。

DW_apb_ssi通过以下握手信号与DMA控制器接口。

- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req
- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req

要使能 DW_apb_ssi 上的 DMA 控制器接口，必须写入 DMA 控制寄存器（DMACR）。向 DMACR 寄存器的 TDMAE 位域写入1可使能 DW_apb_ssi 传输握手接口。向 DMACR 寄存器的 RDMAE 位域写入1可使能 DW_apb_ssi 接收握手接口。

[表580 提供了不同 DMA 传输数据级别值的说明。](#)

表580。DMA
传输数据级别
(DMATDL) 解码
值

DMATDL 值	描述
0000_0000	当传输 FIFO 中无数据项时，dma_tx_req 置位。
0000_0001	当传输 FIFO 中数据项少于或等于1时，dma_tx_req 置位。

0000_0010	当传输 FIFO 中数据项少于或等于2时, dma_tx_req 置位。
...	...
0000_1101	当传输 FIFO 中数据项少于或等于13时, dma_tx_req 置位。
0000_1110	当传输 FIFO 中数据项少于或等于14时, dma_tx_req 置位。
0000_1111	当发送 FIFO 中存在 15 个或更少数据条目时, dma_tx_req 被置位

表 581 对不同 DMA 接收数据级别值进行了说明。

表 581。DMA
接收数据级别
(DMARDL) 解码
值

DMARDL 值	描述
0000_0000	当接收 FIFO 中存在一个或多个数据条目时, dma_rx_req 被置位
0000_0001	当接收 FIFO 中存在两个或多个数据条目时, dma_rx_req 被置位
0000_0010	当接收 FIFO 中存在三个或多个数据条目时, dma_rx_req 被置位
...	...
0000_1101	当接收 FIFO 中存在 14 个或多个数据条目时, dma_rx_req 被置位
0000_1110	当接收 FIFO 中存在 15 个或多个数据条目时, dma_rx_req 被置位
0000_1111	当接收 FIFO 中存在 16 个数据条目时, dma_rx_req 被置位

4.10.11.1. 运行概述

作为块流控设备, DMA 控制器由处理器编程设定通过 DW_apb_ssi 传输或接收的数据项数量 (块大小)。

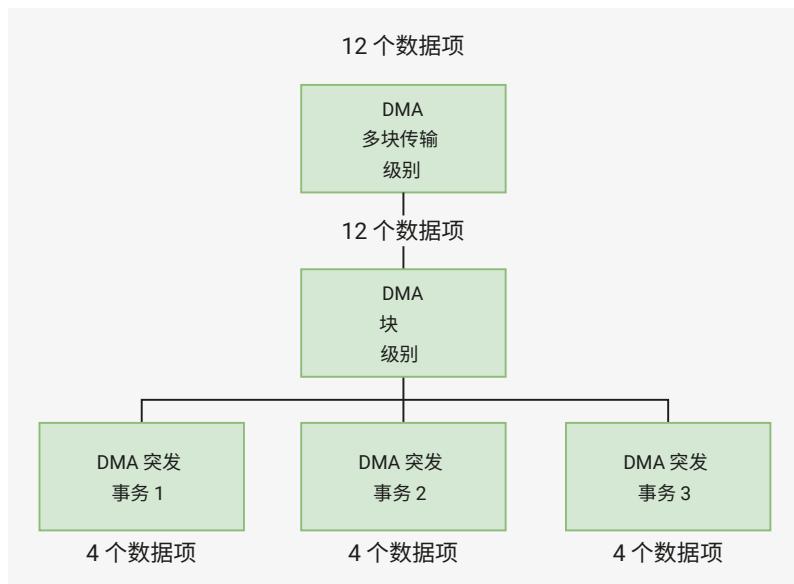
该数据块被划分为多个事务, 每个事务由DW_apb_ssi发起请求。DMA控制器还必须配置每个DMA请求所传输的数据项数量 (本例中为DW_apb_ssi FIFO条目数)。此长度亦称为突发事务长度。

图164展示了单个块传输, DMA控制器中配置的块大小为12, 突发事务长度设置为4。在此情况下, 块大小为突发事务长度的整数倍; 因此, DMA块传输由一系列突发事务组成。

⚠ 注意

在RP2040上, SSI的DMA接口突发事务长度固定为4次传输。SSI.DMARDLR必须始终设为4, 此为复位时的默认值。当SSI的FIFO内有1至3个条目时, SSI将请求单次传输; 当有4个或以上条目时, 则请求4次突发传输。

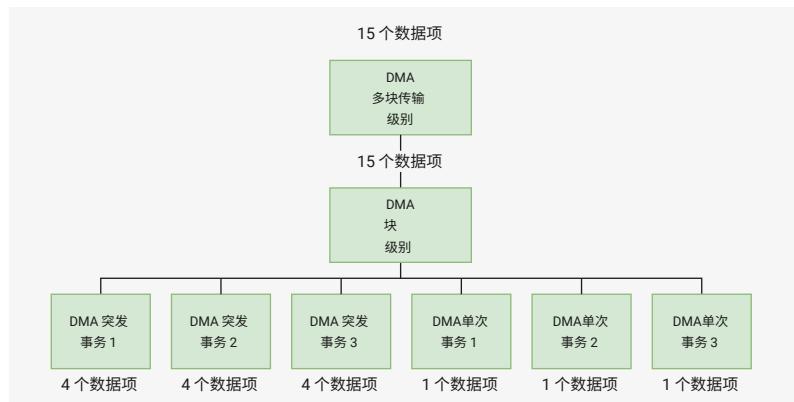
图 164。
DMA 传输拆分为
突发事务。块大
小, DMA.CTLx.BL0
CKS_
TS = 12。每个源突
发事务中的数据
项数量, DMA.CTLx.SR
C_MSIZEx4
SSI 接收 FIFO 水
位线级别, SSI.D
MARDLR+1 = DMA
.CTLx.SRC_MSIZEx4



若 DW_apb_ssi 向该通道发出传输请求，则写入四个数据项至 DW_apb_ssi 传输 FIFO。同理，若 DW_apb_ssi 向该通道发出接收请求，则从 DW_apb_ssi 接收 FIFO 读取四个数据项。须对该 DMA 通道发起三次独立请求，方可完成全部 12 个数据项的写入或读取。

当编程至 DMA 控制器的块大小不是突发事务长度的整数倍时，如图 165 所示，需要通过一系列突发事务及后续单次事务完成块传输。

图 165。
DMA 传输分解为
单次事务和突发
事务。
块大小, DMA.CTLx.BL
OCK_TS = 15。每个
突发事务中的数
据项数量,
DMA.CTLx.DEST_MSIZEx4
= 4. SSI 发送 FIFO
水位线级别,
SSI.DMADLR =
DMA.CTLx.DEST_MSIZEx4



4.10.12. APB 接口

主处理器通过 APB 接口访问 DW_apb_ssi 的数据、控制及状态信息。对 DW_apb_ssi 外设的 APB 访问在以下小节中详细描述。

4.10.12.1. 控制和状态寄存器的 APB 访问

DW_apb_ssi 中的控制与状态寄存器支持字节寻址。DW_apb_ssi 中控制或状态寄存器的最大位宽为 16 位。因此，对 DW_apb_ssi 控制和状态寄存器的所有读写操作仅需一次 APB 访问。

4.10.12.2. 数据寄存器APB访问

DW_apb_ssi中的数据寄存器（DR）宽度为32位，以保持与最大串行传输大小（数据帧）的一致性。对DR进行APB写操作时，数据将从pdata传输至发送FIFO缓冲区。对DR进行APB读操作时，数据将从接收FIFO缓冲区传输至pdata。

DW_apb_ssi的DR寄存器可通过一次APB访问完成写入和读取。

注意

DW_apb_ssi中的DR寄存器在内存映射中占用64个32位位置，以支持AHB突发传输。APB总线本身不支持突发传输，但DW_apb_ssi支持发生在AHB/APB桥接器AHB侧的AHB突发传输。写入上述任一地址位置的操作，效果等同于将数据从pdata总线推入发送FIFO；从上述任一地址读取数据的操作，效果等同于将数据从接收FIFO弹出至pdata总线。DW_apb_ssi上的FIFO缓冲区不可寻址。

4.10.13. 寄存器列表

SSI寄存器起始基址为 `0x18000000`（在SDK中定义为XIP_SSI_BASE）。

表582 SSI寄存器列表

偏移量	名称	说明
0x00	CTRLR0	控制寄存器0
0x04	CTRLR1	主控寄存器1
0x08	SSIENR	SSI使能
0x0c	MWCR	Microwire控制
0x10	SER	从设备使能
0x14	BAUDR	波特率
0x18	TXFTLR	TX FIFO阈值电平
0x1c	RXFTLR	RX FIFO阈值电平
0x20	TXFLR	发射FIFO级别
0x24	RXFLR	接收FIFO级别
0x28	SR	状态寄存器
0x2c	IMR	中断屏蔽
0x30	ISR	中断状态
0x34	RISR	原始中断状态
0x38	TXOICR	发射FIFO溢出中断清除
0x3c	RXOICR	接收FIFO溢出中断清除
0x40	RXUICR	接收FIFO欠流中断清除
0x44	MSTICR	多主中断清除
0x48	ICR	中断清除
0x4c	DMACR	DMA 控制
0x50	DMATDLR	DMA TX 数据级别
0x54	DMARDLR	DMA RX 数据级别

偏移量	名称	说明
0x58	IDR	标识寄存器
0x5c	SSI_VERSION_ID	版本 ID
0x60	DRO	数据寄存器 0 (共36个)
0xf0	RX_SAMPLE_DLY	接收采样延迟
0xf4	SPI_CTRLR0	SPI 控制
0xf8	TXD_DRIVE_EDGE	TX 驱动边缘

SSI：CTRLR0 寄存器

偏移: 0x00

描述

控制寄存器0

表 583。CTRLR0 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	SSTE : 从选择切换使能	读写	0x0
23	保留。	-	-
22:21	SPI_FRF : SPI 帧格式	读写	0x0
	枚举值:		
	0x0 → STD: 标准 1 位 SPI 帧格式; 每个 SCK 1 位, 全双工		
	0x1 → DUAL: 双重 SPI 帧格式; 每个 SCK 两位, 半双工		
	0x2 → QUAD: 四重 SPI 帧格式; 每个 SCK 四位, 半双工		
20:16	DFS_32 : 32 位传输模式下的数据帧大小 n 值 → 每帧 n+1 个时钟	读写	0x00
15:12	CFS : 控制帧大小 n 值 → 每帧 n+1 个时钟	读写	0x0
11	SRL : 移位寄存器环 (测试模式)	读写	0x0
10	SLV_OE : 从机输出使能	读写	0x0
9:8	TMOD : 传输模式	读写	0x0
	枚举值:		
	0x0 → TX_AND_RX: 同时发射和接收		
	0x1 → TX_ONLY: 仅发射 (不适用于 FRF == 0, 即标准 SPI 模式)		
	0x2 → RX_ONLY: 仅接收 (不适用于 FRF == 0, 即标准 SPI 模式)		
	0x3 → EEPROM_READ: EEPROM 读取模式 (先发送 TX, 后接收 RX; RX 于控制数据 TX 完成后开始)		
7	SCPOL : 串行时钟极性	读写	0x0
6	SCPH : 串行时钟相位	读写	0x0
5:4	FRF : 帧格式	读写	0x0
3:0	DFS : 数据帧大小	读写	0x0

SSI: CTRLR1寄存器

偏移: 0x04

描述

主控寄存器1

表584. CTRLR1寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	NDF : 数据帧数量	读写	0x0000

SSI: SSIENR寄存器

偏移: 0x08

描述

SSI使能

表585. SSIENR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	SSI_EN : SSI使能	读写	0x0

SSI: MWCR寄存器

偏移: 0x0c

描述

Microwire控制

表586. MWCR寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	MHS : Microwire握手	读写	0x0
1	MDD : Microwire控制	读写	0x0
0	MWMOD : Microwire传输模式	读写	0x0

SSI: SER寄存器

偏移: 0x10

描述

从属使能

表587. SER寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	针对每个位: 0 → 未选中从属 1 → 选中从属	读写	0x0

SSI: BAUDR 寄存器

偏移: 0x14

描述

波特率

表 588. BAUDR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	SCKDV : SSI 时钟分频器	读写	0x0000

SSI: TXFTLR 寄存器

偏移: 0x18

描述

TX FIFO 阈值电平

表 589. TXFTLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TFT : 发送 FIFO 阈值	读写	0x00

SSI: RXFTLR 寄存器

偏移量: 0x1c

描述

RX FIFO 阈值电平

表 590. RXFTLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RFT : 接收 FIFO 阈值	读写	0x00

SSI: TXFLR 寄存器

偏移: 0x20

描述

发送 FIFO 水平

表 591. TXFLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TFTFL : 发送FIFO级别	只读	0x00

SSI: RXFLR 寄存器

偏移: 0x24

描述

接收FIFO级别

表 592. RXFLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RXTFL : 接收FIFO级别	只读	0x00

SSI: SR 寄存器

偏移: 0x28

描述

状态寄存器

表593。SR寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6	DCOL : 数据冲突错误	只读	0x0
5	TXE : 传输错误	只读	0x0
4	RFF : 接收FIFO满	只读	0x0
3	RFNE : 接收FIFO非空	只读	0x0
2	TFE : 发送FIFO空	只读	0x0
1	TFNF : 发送FIFO未满	只读	0x0
0	BUSY : SSI忙标志	只读	0x0

SSI：IMR寄存器

偏移: 0x2c

描述

中断屏蔽

表594。IMR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIM : 多主控争用中断屏蔽	读写	0x0
4	RXFIM : 接收FIFO满中断屏蔽	读写	0x0
3	RXOIM : 接收FIFO溢出中断屏蔽	读写	0x0
2	RXUIM : 接收FIFO欠载中断屏蔽	读写	0x0
1	TXOIM : 发送FIFO溢出中断屏蔽	读写	0x0
0	TXEIM : 发送FIFO空中断屏蔽	读写	0x0

SSI：ISR寄存器

偏移: 0x30

描述

中断状态

表595。ISR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIS : 多主控争用中断状态	只读	0x0
4	RXFIS : 接收FIFO满中断状态	只读	0x0
3	RXOIS : 接收FIFO溢出中断状态	只读	0x0
2	RXUIS : 接收FIFO欠载中断状态	只读	0x0
1	TXOIS : 发送FIFO溢出中断状态	只读	0x0
0	TXEIS : 发送FIFO空中断状态	只读	0x0

SSI：RISR寄存器

偏移: 0x34

描述

原始中断状态

表596. RISR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIR : 多主控争用原始中断状态	只读	0x0
4	RXFIR : 接收FIFO已满原始中断状态	只读	0x0
3	RXOIR : 接收FIFO溢出原始中断状态	只读	0x0
2	RXUIR : 接收FIFO欠载原始中断状态	只读	0x0
1	TXOIR : 发送FIFO溢出原始中断状态	只读	0x0
0	TXEIR : 发送FIFO为空原始中断状态	只读	0x0

SSI：TXOICR寄存器

偏移: 0x38

描述

发射FIFO溢出中断清除

表597. TXOICR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除发送FIFO溢出中断	只读	0x0

SSI：RXOICR寄存器

偏移: 0x3c

描述

接收FIFO溢出中断清除

表598. RXOICR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除接收FIFO溢出中断	只读	0x0

SSI：RXUICR寄存器

偏移: 0x40

描述

接收FIFO欠流中断清除

表599. RXUICR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除接收FIFO欠载中断	只读	0x0

SSI：MSTICR寄存器

偏移: 0x44

描述

多主中断清除

表600. MSTICR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除多主控争用中断	只读	0x0

SSI：ICR 寄存器

偏移: 0x48

描述

中断清除

表601. ICR寄
存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读取时清除所有激活的中断	只读	0x0

SSI：DMACR 寄存器

偏移: 0x4c

描述

DMA 控制

表602. DMACR寄
存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TDMAE ：传输 DMA 使能	读写	0x0
0	RDMAE ：接收 DMA 使能	读写	0x0

SSI：DMATDLR 寄存器

偏移: 0x50

说明

DMA TX 数据级别

表603. DMATDLR寄
存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	DMATDL ：传输数据水位线	读写	0x00

SSI：DMARDLR 寄存器

偏移: 0x54

描述

DMA 接收数据级别

表604. DMARDLR寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	DMARDL : 接收数据水位线 (DMARDLR+1)	读写	0x00

SSI: IDR 寄存器

偏移: 0x58

描述

识别寄存器

表 605. IDR 寄存器

位	描述	类型	复位值
31:0	IDCODE : 外设标识码	只读	0x51535049

SSI: SSI_VERSION_ID 寄存器

偏移: 0x5c

描述

版本标识

表 606. SSL_VERSION_ID 寄存器

位	描述	类型	复位值
31:0	SSI_COMP_VERSION : SNPS 组件版本 (格式 X.YY)	只读	0x3430312a

SSI: DRO 寄存器

偏移: 0x60

描述

数据寄存器 0 (共36个)

表 607. DRO 寄存器

位	描述	类型	复位值
31:0	DR : 36 个数据寄存器中的第一个	读写	0x00000000

SSI: RX_SAMPLE_DLY 寄存器

偏移: 0xf0

描述

RX 采样延迟

表 608. RX_SAMPLE_DLY 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RSD : RXD 采样延迟 (SCLK 周期)	读写	0x00

SSI: SPI_CTRLR0 寄存器

偏移: 0xf4

描述

SPI控制

表609。
SPI_CTRLR0寄存器

位	描述	类型	复位值
31:24	XIP_CMD: XIP模式下发送的SPI命令（INST_L = 8位）或附加至地址的命令（INST_L = 0位）	读写	0x03
23:19	保留。	-	-
18	SPI_RXDS_EN: 读数据触发使能	读写	0x0
17	INST_DDR_EN: 指令DDR传输使能	读写	0x0
16	SPI_DDR_EN: SPI DDR传输使能	读写	0x0
15:11	WAIT_CYCLES: 控制帧传输与数据接收之间的等待周期（SCLK周期）	读写	0x00
10	保留。	-	-
9:8	INST_L: 指令长度（0/4/8/16位）	读写	0x0
	枚举值：		
	0x0 → NONE：无指令		
	0x1 → 4B：4位指令		
	0x2 → 8B：8位指令		
	0x3 → 16B：16位指令		
7:6	保留。	-	-
5:2	ADDR_L: 地址长度（以4位递增，范围0b-60b）	读写	0x0
1:0	TRANS_TYPE: 地址与指令传输格式	读写	0x0
	枚举值：		
	0x0 → 1C1A：命令和地址均采用标准SPI帧格式		
	0x1 → 1C2A：命令采用标准SPI格式，地址采用指定格式由FRF指定		
	0x2 → 2C2A：命令和地址均采用FRF指定的格式（例如双SPI）		

SSI: TXD_DRIVE_EDGE寄存器

偏移: 0xf8

说明

TX驱动边沿

表610。
TXD_DRIVE_EDGE
寄存器

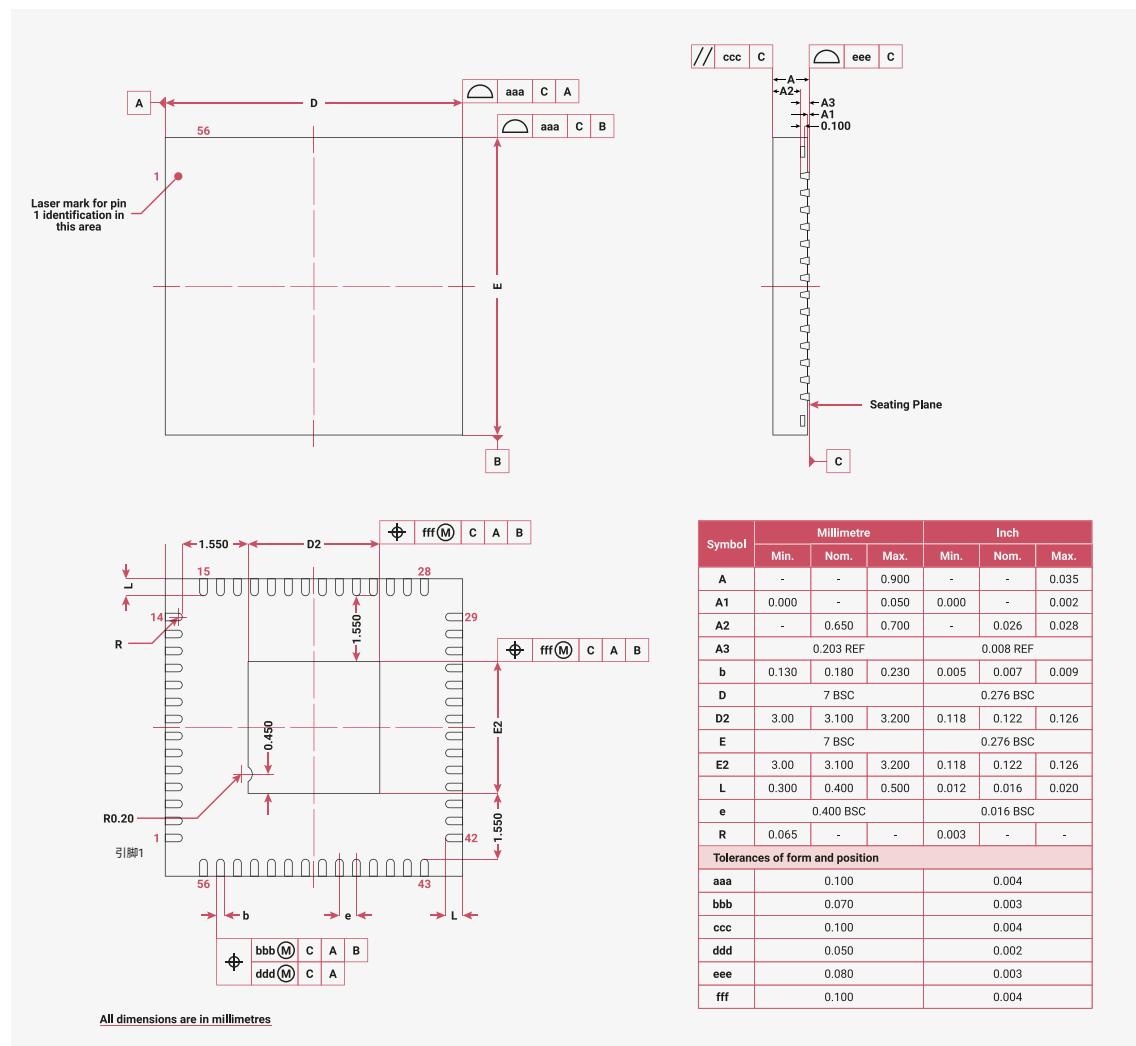
位	描述	类型	复位值
31:8	保留。	-	-
7:0	TDE: TXD 驱动边沿	读写	0x00

第5章。电气与机械

RP2040芯片的物理与电气细节。

5.1. 封装

图166。RP2040 QFN-56封装的俯视图（左上）、侧视图（右上）及底视图（左下）



注意

QFN封装中的中央接地焊盘（或ePad）无统一标准尺寸，然而RP2040上的尺寸小于多数封装。这意味着附带CAD工具的标准0.4mm QFN-56焊盘图可能需进行调整。此设计允许在中央焊盘与外围焊盘间布线，有助于维护较低成本PCB上的电源及接地完整性。详见最小设计示例。

● 注意

引脚涂覆哑光锡（Sn）表面。退火在电镀后进行，条件为150°C，持续1小时。引脚镀层最低厚度为8微米，中间层材料为CuFe2P（粗化铜，Cu）。

5.1.1. 热特性

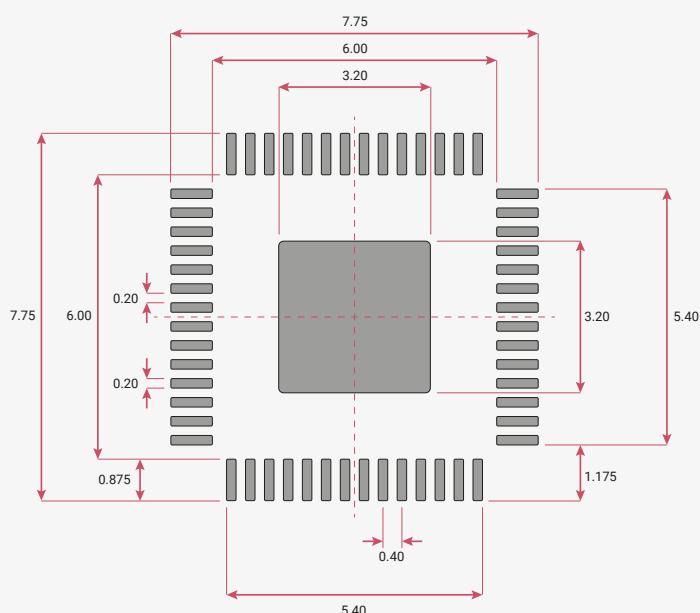
封装热性能数据详见表611。

表611。RP2040 QFN 56封装的热性能数据。

θ_{JA} (°C/W)	Ψ_{JT} (°C/W)	Ψ_{JB} (°C/W)	T_J (°C)	T_T (°C)	θ_{JC} (°C/W)	θ_{JB} (°C/W)
48.00	0.80	29.20	42.00	41.8	19.01	29.03

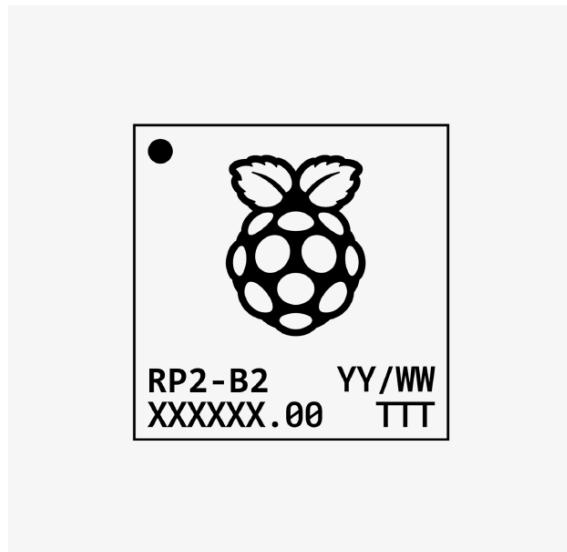
5.1.2. 推荐 PCB 布局

图 167。
RP2040 QFN-56 封装的推荐 PCB 封装尺寸



5.1.3. 封装标记

RP2040 7x7 毫米 QFN-56 封装如图 168 所示，规格见表 612。
坐标原点位于封装左下角。

图 168。封装
标记格式表 612。标记
要求及
尺寸

线	步骤	项目	X 坐标	坐标 Y	字符高度	字符宽度	字符间距
1	1	引脚1点	0.5	6	0.5	0.5	
2	1	标识	3.5	2.395	3.83	3.05	
3	1	RP2-B2	0.555	1.585	0.61	0.37	0.09
3	2	YY/WW	4.235	1.585	0.61	0.37	0.09
4	1	XXXXXX.00	0.555	0.775	0.61	0.37	0.09
4	2	TTT (可选)	5.155	0.775	0.61	0.37	0.09

① 注意

在第3行第1步中，“RP2-B2”标记表示器件名称“RP2”及硅片版本“B2”。

5.2. 存储条件

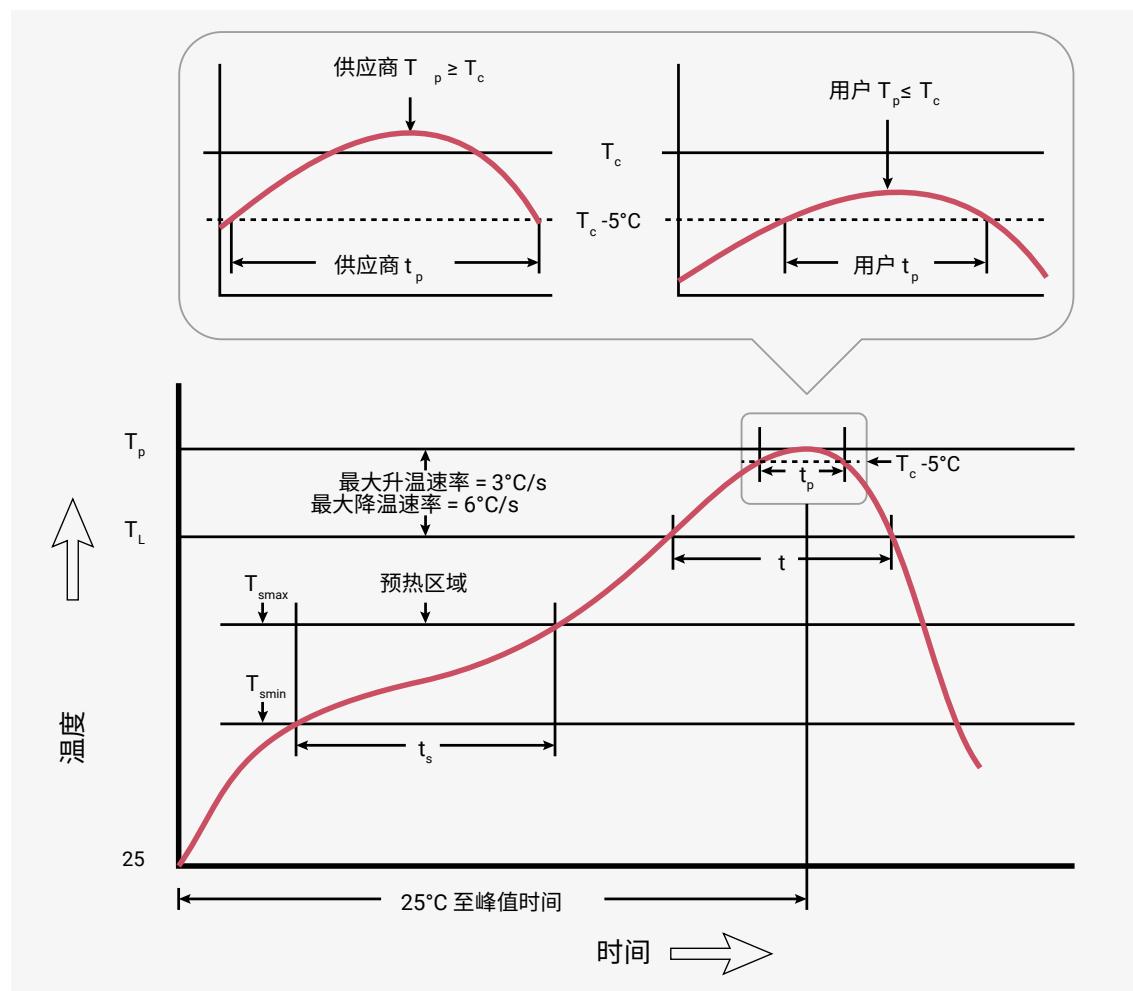
为保持裸露RP2040器件的货架期及存放寿命，建议依照J-STD（020E及033D）针对RP2040（归类为MSL1）的储存条件，保持温度低于30°C、相对湿度不超过85%。

5.3. 焊接曲线

RP2040为无铅器件，其 T_p 值为260°C。

所有温度均指封装中心，测量于组装返修时朝上的封装主体表面（live-bug方向）。若器件以非正常live-bug返修方向（即dead-bug）进行返修， T_p 应控制在live-bug $T_p \pm 2^\circ\text{C}$ 内，且仍需满足 T_c 要求；否则，应调整曲线以实现后者。

图169。
分类轮廓
(非按比例绘制)



① 注意

本文件中的回流轮廓用于分类/预处理，不应用于规定电路板组装轮廓。实际电路板组装轮廓应依据具体工艺需求和电路板设计制定，且不得超过表613中的参数。

表613。焊接轮廓数值

轮廓特征	数值
最低温度 (T_{smin})	150°C
最高温度 (T_{smax})	200°C
从 (T_{smin} 至 T_{smax}) 的时间 (t_s)	60–120秒
升温速率 (从 T_L 至 T_p)	最大3°C/秒
液相温度 (T_L)	217°C
保持高于 T_L 的时间 (t_L)	60 到 150 秒
峰值封装体温 (T_p)	260°C
分类温度 (T_c)	260°C
在规定分类温度 (T_c) $\pm 5^\circ\text{C}$ 范围内的时间 (t_p)	30 秒
降温速率 (从 T_p 到 T_L)	最大 6°C/秒
从 25°C 到峰值温度的时间	最长 8 分钟

5.4. 合规性

RP2040 符合湿度敏感等级 1 的要求。

RP2040 符合欧洲化学品管理署（ECHA）于2020年6月25日公布的高度关注物质（SVHC）REACH要求。

RP2040 符合RoHS指令（欧盟）2011/65/EU及指令（欧盟）2015/863中有关受控环境相关物质的标准及要求。

RP2040所进行的封装级可靠性鉴定：

- 温度循环测试，依据 JESD22-A104 标准
- 高加速应力测试（HAST），依据 JESD22-A110 标准
- 高温储存寿命测试（HTSL），依据 JESD22-A103 标准

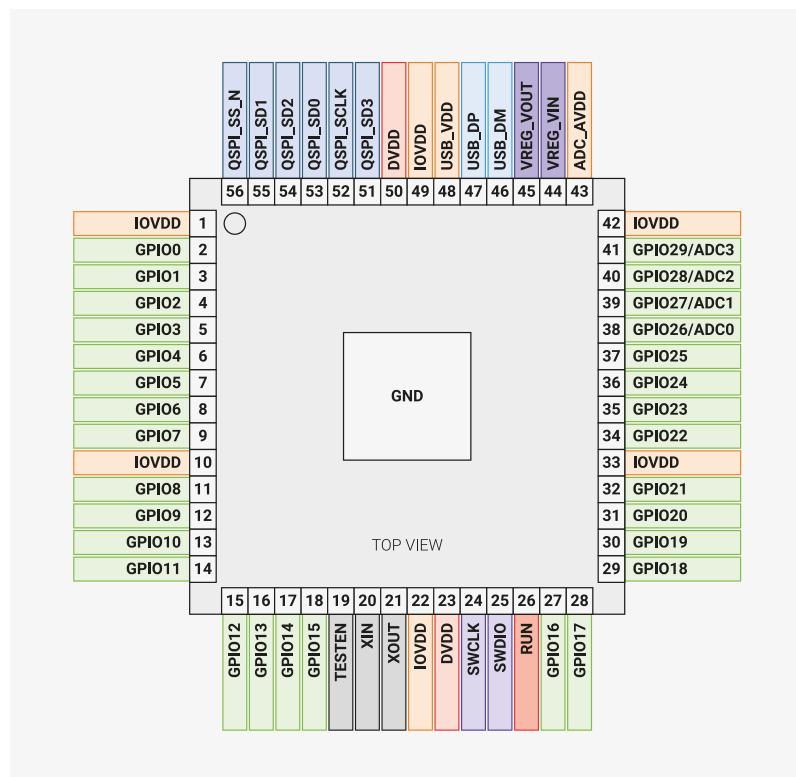
i 注意

由于RP2040为底部终端器件（QFN封装），不适用JEDEC标准（JESD201A），因此未进行锡须测试。

5.5. 引脚定义

5.5.1. 引脚位置

图170. RP2040
QFN-56封装
引脚排列



5.5.2. 引脚说明

5.5.2.1. 引脚类型

下表GPIO引脚（表615）中，引脚类型定义如下。

表614. 引脚类型

引脚类型	方向	描述
数字输入	仅输入	标准数字。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
数字输入输出	双向	
数字输入（容错）	仅输入	容错数字。这些引脚被描述为容错型，即当引脚电压低于3.63V且IOVDD为0V时，流入该引脚的电流极小。这些引脚还具备增强的ESD保护功能。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
数字IO (FT)	双向	
数字IO / 模拟	双向（数字），输入（模拟）	标准数字及ADC输入。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
USB IO	双向	这些引脚用于USB功能，依据USB规范内含内部上拉及下拉电阻。注意USB操作需外部27Ω串联电阻。
模拟(XOSC)		用于连接12MHz晶体的振荡器输入引脚。或者，XIN端可由方波信号驱动。

5.5.2.2. 引脚列表

表615. GPIO引脚

名称	编号	类型	电源域	复位状态	描述
GPIO0	2	数字IO (FT)	IOVDD	下拉	用户IO
GPIO1	3	数字IO (FT)	IOVDD	下拉	用户IO
GPIO2	4	数字IO (FT)	IOVDD	下拉	用户IO
GPIO3	5	数字IO (FT)	IOVDD	下拉	用户IO
GPIO4	6	数字IO (FT)	IOVDD	下拉	用户IO
GPIO5	7	数字IO (FT)	IOVDD	下拉	用户IO
GPIO6	8	数字IO (FT)	IOVDD	下拉	用户IO
GPIO7	9	数字IO (FT)	IOVDD	下拉	用户IO
GPIO8	11	数字IO (FT)	IOVDD	下拉	用户IO
GPIO9	12	数字IO (FT)	IOVDD	下拉	用户IO
GPIO10	13	数字IO (FT)	IOVDD	下拉	用户IO
GPIO11	14	数字IO (FT)	IOVDD	下拉	用户IO
GPIO12	15	数字IO (FT)	IOVDD	下拉	用户IO
GPIO13	16	数字IO (FT)	IOVDD	下拉	用户IO
GPIO14	17	数字IO (FT)	IOVDD	下拉	用户IO
GPIO15	18	数字IO (FT)	IOVDD	下拉	用户IO

名称	编号	类型	电源域	复位状态	描述
GPIO16	27	数字IO (FT)	IOVDD	下拉	用户IO
GPIO17	28	数字IO (FT)	IOVDD	下拉	用户IO
GPIO18	29	数字IO (FT)	IOVDD	下拉	用户IO
GPIO19	30	数字IO (FT)	IOVDD	下拉	用户IO
GPIO20	31	数字IO (FT)	IOVDD	下拉	用户IO
GPIO21	32	数字IO (FT)	IOVDD	下拉	用户IO
GPIO22	34	数字IO (FT)	IOVDD	下拉	用户IO
GPIO23	35	数字IO (FT)	IOVDD	下拉	用户IO
GPIO24	36	数字IO (FT)	IOVDD	下拉	用户IO
GPIO25	37	数字IO (FT)	IOVDD	下拉	用户IO
GPIO26 / ADC0	38	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO27 / ADC1	39	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO28 / ADC2	40	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO29 / ADC3	41	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入

表 616. QSPI 引脚

名称	编号	类型	电源域	复位状态	描述
QSPI_SD3	51	数字输入输出	IOVDD		QSPI 数据
QSPI_SCLK	52	数字输入输出	IOVDD	下拉	QSPI 时钟
QSPI_SD0	53	数字输入输出	IOVDD		QSPI 数据
QSPI_SD2	54	数字输入输出	IOVDD		QSPI 数据
QSPI_SD1	55	数字输入输出	IOVDD		QSPI 数据
QSPI_CS _n	56	数字输入输出	IOVDD	上拉	QSPI 片选

表 617. 晶体振荡器引脚

名称	编号	类型	电源域	描述
XIN	20	模拟 (XOSC)	IOVDD	晶体振荡器。XIN 也可由方波驱动。
XOUT	21	模拟 (XOSC)	IOVDD	晶体振荡器。

表 618. 串行线调试引脚

名称	编号	类型	电源域	复位状态	描述
SWCLK	24	数字输入 (容错)	IOVDD	上拉	调试时钟
SWD	25	数字IO (FT)	IOVDD	上拉	调试数据

表 619. 杂项引脚

名称	编号	类型	电源域	复位状态	描述
RUN	26	数字输入 (容错)	IOVDD	上拉	芯片使能 / 复位

名称	编号	类型	电源域	复位状态	描述
测试	19	数字输入	IOVDD	下拉	测试使能（连接至接地）

表 620. USB 引脚

名称	编号	类型	电源域	描述
USB_DP	47	USB IO	USB_VDD	USB 数据正极，USB 运行需串联27Ω电阻
USB_DM	46	USB IO	USB_VDD	USB 数据负极，USB 运行需串联27Ω电阻

表 621. 电源引脚

名称	编号	描述
IOVDD	1, 10, 22, 33, 42, 49	IO 电源
DVDD	23, 50	核心电源
VREG_VIN	44	稳压器输入电源
VREG_VOUT	45	稳压器输出
USB_VDD	48	USB 电源
ADC_AVDD	43	ADC 电源
GND	57	通过中心焊盘连接公共地线

5.5.3. 引脚规格

以下电气规格基于指定温度和电压范围内的特性测试及工艺变异获得，除非规格标注为“模拟”。在此情况下，数据仅供参考，且不保证其准确性。

5.5.3.1. 绝对最大额定值

表 622。 数字IO（标准与容错）绝对最大额定值

参数	符号	最小值	最大值	单位	备注
I/O 供电电压	IOVDD	-0.5	3.63	V	
I/O端电压	V _{引脚}	-0.5	IOVDD + 0.5	V	

5.5.3.2. 静电放电性能

表 623。除非另有说明，所有引脚的静电放电性能

参数	符号	最大值	单位	备注
人体模型	HBM	2	kV	符合 JEDEC 规格 JS-01-2012 (2012年4月)

参数	符号	最大值	单位	备注
人体模型 仅适用于数字 (FT) 引脚	HBM	4	kV	符合 JEDEC 规格 JS-0 01-2012 (2012年4 月)
带电器件模型	CDM	500	V	符合 JESD22-C101E (2009年12月)

5.5.3.3. 热性能

表624. 热性能

参数	符号	最小值	典型值	最大值	单位	备注
封装 温度	T _C	-40		85	°C	

5.5.3.4. IO电气特性

表625. 数字IO特性
- 标准及FT,
除非另有说明

参数	符号	最小值	最大值	单位	备注
引脚输入漏电流	I _{IN}		1	μA	
输入高电压 @ IOVDD=1.8V	V _{IH}	0.65 * IOVDD	IOVDD + 0.3	V	
输入高电压 @ IOVDD=2.5V	V _{IH}	1.7	IOVDD + 0.3	V	
输入高电压 @ IOVDD=3.3V	V _{IH}	2	IOVDD + 0.3	V	
输入电压低 @ IOVDD=1.8V	V _{IL}	-0.3	0.35 * IOVDD	V	
输入电压低 @ IOVDD=2.5V	V _{IL}	-0.3	0.7	V	
输入电压低 @ IOVDD=3.3V	V _{IL}	-0.3	0.8	V	
输入迟滞 电压 @ IOVDD=1.8V	V _{HYS}	0.1 * IOVDD		V	施密特触发器 已启用
输入迟滞 电压 @ IOVDD=2.5V	V _{HYS}	0.2		V	施密特触发器 已启用
输入迟滞 电压 @ IOVDD=3.3V	V _{HYS}	0.2		V	施密特触发器 已启用
输出电压 高电平 @ IOVDD=1.8V	V _{OH}	1.24	IOVDD	V	I _{OH} = 2、4、8 或12mA，取决 于设置

参数	符号	最小值	最大值	单位	备注
输出电压 高电平 @ $I_{OVDD}=2.5V$	V_{OH}	1.78	I_{OVDD}	V	$I_{OH}=2、4、8$ 或12mA，取决于设置
输出电压 高电平 @ $I_{OVDD}=3.3V$	V_{OH}	2.62	I_{OVDD}	V	$I_{OH}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=1.8V$	V_{OL}	0	0.3	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=2.5V$	V_{OL}	0	0.4	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=3.3V$	V_{OL}	0	0.5	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
上拉电阻	R_{PU}	50	80	kΩ	
下拉 电阻	R_{PD}	50	80	kΩ	
最大总 I_{OVDD} 电流	$I_{I_{OVDD_MAX}}$		50	mA	GPIO 与 QSPI 引脚所提供电流之总和
最大总 由 $IO (I_{OVSS})$ 引起的 VSS 电流	$I_{I_{OVSS_MAX}}$		50	mA	流入 GPIO 与 QSPI 引脚的电流总和

表 626. USB IO 特性

参数	符号	最小值	最大值	单位	备注
引脚输入漏电流	I_{IN}		1	μA	
单端输入 高电压	V_{IHSE}	2		V	
单端输入 低电压	V_{ILSE}		0.8	V	
差分输入 高电压	V_{IHDIFF}	0.2		V	
差分输入 低电压	V_{ILDIFF}		-0.2	V	
输出电压 高	V_{OH}	2.8	USB_VDD	V	
输出电压 低	V_{OL}	0	0.3	V	
上拉电阻 - R_{PU2}	R_{PU2}	0.873	1.548	kΩ	

参数	符号	最小值	最大值	单位	备注
上拉电阻 — R _{P1&2}	R _{P1&2}	1.398	3.063	kΩ	
下拉 电阻	R _{P0}	14.25	15.75	kΩ	

表627. ADC
特性

参数	符号	最小值	最大值	单位	备注
ADC输入电压 范围	V _{PIN_ADC}	0	ADC_AVDD	V	
有效位数	ENOB	8.7		位	参见第4.9.3节
分辨位			12	位	
ADC输入 阻抗	R _{IN_ADC}	100		kΩ	

表628。使用方波
输入时的振荡器
引脚特性

参数	符号	最小值	最大值	单位	备注
输入高电压	V _{IH}	0.65*IOVDD	IOVDD + 0.3	V	仅XIN。XOUT 悬空
输入电压低	V _{IL}	0	0.35*IOVDD	V	仅XIN。XOUT 悬空

有关振荡器的详细信息，请参见第2.16节；有关晶体使用的信息，请参见最小设计示例。

5.5.3.5. GPIO输出电压规格的解释

RP2040上的GPIO具有四种不同的输出驱动强度，名义上分别称为2、4、8和12mA模式。

这并非硬性限制，也不意味着GPIO始终会提供（或吸收）所选的电流强度。GPIO提供或吸收的电流大小取决于所连接的负载。GPIO会尝试将输出驱动至IOVDD电平（逻辑0时为0V），但其可提供的电流受所选驱动强度限制。因此，电流负载越大，针脚处的电压越低。在某个临界点，GPIO输出的电流将非常大，导致电压降至无法被连接设备输入识别为逻辑1。表625中的输出规格旨在量化在从针脚拉取规定电流时，电压可能降低的幅度。

输出高电压（V_{OH}）定义为在特定选定驱动强度下，输出针脚驱动至逻辑1时的最低电压；例如，在4mA驱动强度模式下，针脚源出4mA电流。输出低电压的定义类似，但指针脚驱动逻辑0时的最低电压。

此外，从IOVDD电源组（即GPIO和QSPI针脚）输出的所有IO电流总和（当输出为高电平时）不得超过I_{IOVDD_MAX}。同样，所有被下拉吸收的IO电流总和（即输出被拉低时）不得超过I_{IOVSS_MAX}。

图171。GPIO输出的典型电流-电压曲线。

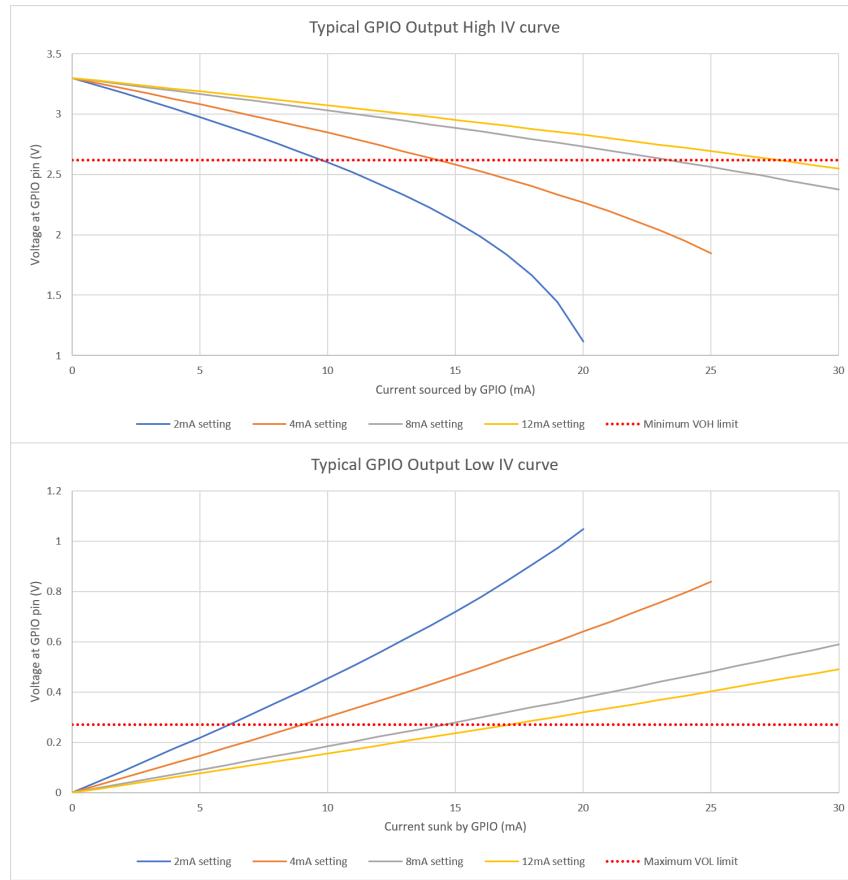


图171显示了随着引脚负载电流增加，对输出电压的影响。您可以清楚地观察不同驱动强度的影响；驱动强度越大，在给定电流下，输出电压越接近 IOVDD（或 0V）。最小 V_{OH} 和最大 V_{OL} 极限以红色标出。您可以看到，在每种驱动强度规定的电流下，电压均在许可极限之内，这意味着该特定器件能够驱动更大电流，且仍符合 V_{OH}/V_{OL} 规格。这是室温下的典型器件，其他器件的电压分布可能更接近该极限。当然，如果您的应用不需要如此严格控制的电压，则GPIO可以源出或吸入超过所选驱动强度设置的电流，但需要通过实验确定在您的应用中是否确实安全，因为这将超出本规范的范围。

5.5.3.6. 引脚IO延迟

这些延迟包括PIO的输入/输出映射逻辑、IO多路复用及实际垫片在标称负载5 pF下的延迟。最大/最小值涵盖工艺变化、电压（ $1.1\text{ V} \pm 10\%$ ）和温度（ -40°C 至 125°C ）的极限条件。

这些延迟假设IOVDD为1.8 V，且 [PADS_VSEL](#)已设置。IOVDD为3.3 V时，延迟显著降低且范围更小。

触发器本身的典型建立时间为10.6 ps，保持时间为2.2 ps。必须考虑触发器与垫片之间的IO延迟。

有关最小和最大输出延迟，从 [CLK_SYS](#)信号达到PIO中任一触发器至特定GPIO引脚数据有效的时间，详见表629。

表629。引脚从触发器到引脚的最小和最大延迟，单位：纳秒。

引脚输出	最小延迟（纳秒）	最大延迟（纳秒）
GPIO0	2.27	7.10
GPIO1	2.31	7.07

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO2	2.33	7.08
GPIO3	2.24	7.00
GPIO4	2.30	7.07
GPIO5	2.34	7.10
GPIO6	2.32	7.10
GPIO7	2.39	7.09
GPIO8	2.34	7.09
GPIO9	2.38	7.08
GPIO10	2.33	7.07
GPIO11	2.36	7.08
GPIO12	2.35	7.04
GPIO13	2.31	7.08
GPIO14	2.38	7.06
GPIO15	2.33	7.05
GPIO16	2.34	7.09
GPIO17	2.37	7.09
GPIO18	2.37	7.04
GPIO19	2.27	7.10
GPIO20	2.38	7.09
GPIO21	2.05	7.10
GPIO22	2.34	7.07
GPIO23	2.16	7.05
GPIO24	2.12	7.06
GPIO25	2.26	7.10
GPIO26	2.32	7.09
GPIO27	2.26	7.08
GPIO28	2.34	7.09
GPIO29	2.30	7.07

有关最小和最大输入延迟，从引脚输入到输入同步器，详见表630。

表630。引脚
从引脚输入
到输入同步器的最
小和最大延迟
，单位：纳秒
。

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO1	1.89	5.22
GPIO2	1.84	5.25
GPIO3	1.83	5.24
GPIO4	1.90	5.17
GPIO5	1.90	5.14

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO6	1.91	5.19
GPIO7	1.91	5.14
GPIO8	1.95	5.14
GPIO9	1.96	5.12
GPIO10	1.95	5.11
GPIO11	1.92	5.16
GPIO12	1.92	5.15
GPIO13	1.94	5.16
GPIO14	1.90	5.18
GPIO15	1.92	5.15
GPIO16	1.95	5.13
GPIO17	1.95	5.12
GPIO18	1.95	5.10
GPIO19	1.95	5.12
GPIO21	2.07	4.98
GPIO23	1.98	5.06
GPIO24	1.97	5.07
GPIO25	1.97	5.08
GPIO26	1.96	5.12
GPIO27	1.94	5.13
GPIO28	1.95	5.13
GPIO29	1.99	5.10

有关在所有工况下，从引脚输入到状态机IN数据触发器（绕过同步器）的最小和最大输入延迟，详见表631。

表631。引脚
从引脚输入
到状态机IN数据触
发器（绕过同
步器）的最小和最
大延迟，单
位：纳秒。

引脚输入	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO1	2.22	5.45
GPIO2	2.25	5.49
GPIO3	2.23	5.18
GPIO4	2.24	5.41
GPIO5	2.30	5.65
GPIO6	2.25	5.48
GPIO7	2.26	5.50
GPIO8	2.30	5.51
GPIO9	2.25	5.68
GPIO10	2.34	5.71
GPIO11	2.28	5.47

引脚输入	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO12	2.29	5.40
GPIO13	2.25	5.47
GPIO14	2.24	5.41
GPIO15	2.23	5.47
GPIO16	2.30	5.42
GPIO17	2.28	5.44
GPIO18	2.28	5.34
GPIO19	2.30	5.50
GPIO21	2.16	5.79
GPIO23	2.33	5.53
GPIO24	2.28	5.60
GPIO25	2.29	5.53
GPIO26	2.28	5.38
GPIO27	2.27	5.39
GPIO28	2.24	5.28
GPIO29	2.33	5.47

5.5.3.6.1. IOVDD的影响

上述所有IO延迟均基于 $\text{IOVDD} = 1.8\text{V}$ 的假设。提高 IOVDD 至 3.3V 将显著降低引脚延迟，而引脚延迟在上述延迟中占有较大比重。请参见表 632，了解在 1.8V 和 3.3V 下最佳与最差引脚延迟的汇总。

表 632。 1.8V 和 3.3V 下的最佳与最差引脚延迟。

路径类型	IOVDD	最小延迟 (纳秒)	最大延迟 (纳秒)
输出	1.8V	1.54	3.65
输出	3.3V	1.11	2.14
输入	1.8V	0.63	1.06
输入	3.3V	0.47	0.76

更改 IOVDD 不影响核心域内的任何逻辑，因此可将这些差异加至上述IO延迟表中，以估算 $\text{IOVDD} = 3.3\text{V}$ 时的IO延迟范围（详见表 633）。

表 633。 IO 延迟
 IO 延迟范围， $\text{IOVDD} = 3.3\text{V}$ 时。

路径组	IOVDD	最小延迟 (纳秒)	最大延迟 (纳秒)
输出	1.8V	2.12	7.10
输出	3.3V	1.69	5.59
输入至同步	1.8V	1.83	5.25
输入至同步	3.3V	1.67	4.95
输入至 SM	1.8V	2.16	5.79
输入至 SM	3.3V	2.00	5.49

5.6. 电源供应

表 634。电源规格

电源	电源供应	最小值	典型值	最大值	单位
IOVDD ^a	数字输入输出	1.62	1.8 / 3.3	3.63	V
DVDD ^b	数字核心	1.05	1.1	1.16	V
VREG_VIN	电压调节器	1.62	1.8 / 3.3	3.63	V
USB_VDD	USB PHY	3.135	3.3	3.63	V
ADC_AVDD ^c	ADC	1.62	3.3	3.63	V

^a 如果 IOVDD < 2.5V，GPIO VOLTAGE_SELECT 寄存器应相应调整。详见第 2.9 节。

^b 短期瞬态应控制在 +/-100mV 以内。如第 2.15.3 节所述使用 200MHz 作为 clk_sys，DVDD 应设为 1.15V。

^c ADC 性能在电压低于 2.97V 时将受损。

5.7. 功耗

5.7.1. 外设功耗

基线读数在 WAKE_EN0/WAKE_EN1 寄存器仅激活时钟源及基本外设 (BUSCTRL、BUSFAB、VREG、复位、ROM、SRAM) 状态下采集。时钟设为默认时钟配置。对于各外设，通过在 WAKE_EN0/WAKE_EN1 寄存器中启用该外设所有时钟源，依次激活。当前电流消耗是启用外设时电流的增量。

表 635。基线功耗

外设	典型 DVDD 电流消耗 ($\mu\text{A}/\text{MHz}$)
DMA	2.6
I2C0	3.9
I2C1	3.8
IO + 垫片	23.6
PIO0	12.3
PIO1	12.5
PWM	5.0
RTC	1.1
SIO	1.9
SPI0	1.7
SPI1	1.8
定时器	1.2
UART0	3.5
UART1	3.7
看门狗	1.0
XIP	37.6

由于固定的 48 MHz 外部参考时钟及可变系统时钟输入，ADC 和 USBCTRL 的功耗不会随着系统时钟线性变化（而其他仅具有系统和/或外设时钟输入的外设则会如此）。在标准时钟（系统时钟 125 MHz）下，ADC 和 USBCTRL 模块的绝对 D VDD 电流消耗如下：

表 636。ADC 和 USBCTRL 的基线功耗

外设	典型 DVDD 电流消耗 ($\mu\text{A}/\text{MHz}$)
ADC	0.1
USBCTRL	1.3

5.7.2. 典型用户案例功耗

以下数据展示了三种典型（ t_t ）、快速（ f_f ）和慢速（ s_s ）RP2040 器件在四种不同软件使用场景下，各电源的电流消耗情况。

i 注意

有关 Raspberry Pi Pico 的功耗详情，请参见[Raspberry Pi Pico 数据手册](#)。

首先，“Popcorn”（媒体播放器演示）使用了VGA、SD卡和音频扩展板。该演示采用VGA视频、I2S音频以及4位SD卡访问，系统时钟频率为48MHz。

i 注意

有关VGA扩展板的更多详情，请参阅《[RP2040硬件设计](#)》一书。

其次，是RP2040的BOOTSEL模式。这些测量分别在总线有无USB活动的情况下进行，使用Raspberry Pi 4作为主机。

第三个用例使用了[hello_dormant](#)二进制文件，该文件使RP2040进入低功耗状态，即休眠模式。

最后一个用例使用了[hello_sleep](#)二进制代码，使RP2040进入低功耗状态，即睡眠模式。

表637为每个电源供应列出了两栏：“典型平均电流”和“最大平均电流”。前者为在室温和标称电压条件下（例如DVDD=1.1V, IOVDD=3.3V等），典型RP2040在数秒内的平均电流。“最大平均电流”指在极端温度和最高电压条件下（例如 DVDD=1.21V 等），[最坏情况的 RP2040 设备上可能出现的最大电流消耗](#)（为数秒的平均值）。

i 注意

“爆米花”电流消耗测量取决于当前显示的视频内容。“典型”值为播放多种颜色和强度视频数秒后的平均电流。“最大”值为播放白色视频期间测得的电流，此时电流需求最高。

表637. 功耗

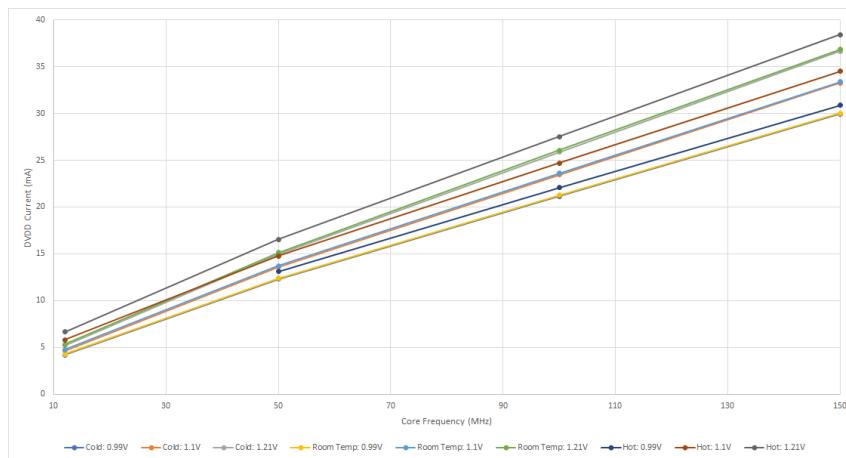
软件使用案例	典型平均 DVDD 电流	最大平均 DVDD 电流	典型平均 IOVDD 电流	最大平均 IOVDD 电流	典型平均 USB_VDD 电流	最大平均 USB_VDD 电流	单位
爆米花	10.9	16.6	24.8	35.5	-	-	mA
BOOTSEL 模式 - 活跃	9.4	14.7	1.2	4.3	1.4	2.0	mA
BOOTSEL 模式 - 空闲	9.0	14.3	1.2	4.3	0.2	0.6	mA
休眠	0.18	4.2	-	-	-	-	mA

软件使用案例	典型 平均 DVDD 电流	最大平均 DVDD 电流	典型 平均 IOVDD 电流	最大平均 IOVDD 电流	典型 平均 USB_VDD 电流	最大平均 USB_VDD 电流	单位
睡眠	0.39	4.5	-	-	-	-	mA

5.7.2.1. 功耗与频率的关系

为表明RP2040核心运行频率与DVDD电源电流消耗之间的关系，图172展示了典型RP2040器件在各核心时钟频率下，双核持续执行FFT计算的测量结果。图172还展示了封装温度及DVDD电压对电流消耗的影响。

图172。典型RP2040器件运行FFT计算时，DVDD电流与核心频率的关系



附录A：寄存器字段类型

标准类型

RW:

- 读/写
- 读操作返回寄存器的值
- 写操作更新寄存器的值

RO:

- 只读
- 读操作返回寄存器的值
- 写操作将被忽略

WO:

- 只写
- 读操作返回0
- 写操作更新寄存器的值

清除类型

SC

- 自清除
- 向SC字段中的位写入1将触发事件，事件触发后该位会自动清零
- 向SC字段中的位写入0不会执行任何操作

WC

- 写-清除
- 向WC字段中的位写入1会将该位写为0
- 向WC字段中的位写入0不产生任何效果
- 读操作返回寄存器的值

FIFO 类型

这些字段用于从 FIFO 读写数据，配套寄存器提供 FIFO 的控制和状态信息。控制和状态寄存器无固定格式，因其针对各 FIFO 接口特定设计。

RWF

- 读/写 FIFO
- 读取该字段返回来自 FIFO 的数据
 - 读取完成后，数据值将从 FIFO 中移除
 - 如 FIFO 为空，将返回默认值；默认值因各 FIFO 接口而异
- 写入此字段的数据将被推入 FIFO，FIFO 满时的行为由各 FIFO 接口具体定义
- 读写操作可能访问不同的 FIFO

RF

- 读 FIFO
- 功能与 RWF 相同，但只读

WF

- 写 FIFO
- 功能与 RWF 相同，但只写

附录 B：勘误

硬件模块按字母顺序排列。勘误按编号列于相关章节下。

Bootrom

RP2040-E9

参考	RP2040-E9
摘要	ROM引导程序无法直接引导进入作为SRAM的XIP缓存
描述	<p>当XIP缓存被禁用时（第2.6.3.1节），XIP缓存可用作额外的16kB SRAM块。UF2引导程序支持仅RAM的UF2二进制文件，直接加载到内存中，并通过看门狗复位进入。单个UF2二进制文件可以初始化XIP缓存内容与主系统内存，引导程序会禁用缓存，以便写入缓存内容。</p> <p>然而，看门狗复位会重新启用缓存，因此直接引导进入作为SRAM的缓存别名会导致立即发生总线错误。缓存内容得到保留，但启动后无法立即访问。</p>
解决方案	<p>在主SRAM中添加代码，在访问作为SRAM的缓存别名之前重新禁用XIP缓存。进入仅RAM的UF2二进制文件时，引导程序选择主SRAM或作为SRAM的缓存中较低的加载地址作为入口点，若两者均已加载，则优先选择主SRAM。</p> <p>此外，如果 <code>0x15...</code> 段在启动后立即写入，则需要对FLUSH寄存器执行虚读，以确保在看门狗触发的标签存储器刷新过程中不会发生缓存即SRAM的写入（参见第2.6.3.2节）。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档

RP2040-E14

参考	RP2040-E14
摘要	稀疏或未对齐的闪存二进制UF2可能无法被UF2引导程序正确写入闪存

描述	<p>RP2040的UF2文件由256字节的数据页组成，每页均标记为由UF2引导程序写入指定地址。闪存二进制UF2是指其每个256字节的数据页均被标记为写入闪存中256字节对齐地址的UF2文件。</p> <p>写入闪存时，必须先擦除整个4kB闪存扇区，随后方可对该扇区内的任意数据页进行（重新）写入。UF2引导加载程序不要求闪存二进制UF2必须包含某个扇区内所有页面的数据。在这种情况下，整个扇区将首先被擦除，随后写入存在的页面，4kB扇区的其余部分将保持未定义状态。</p> <p>当部分填充的扇区位于二进制文件末尾时，此机制能够如预期般正常工作；这非常常见，因为二进制文件不必是4kB的整数倍长度。</p> <p>然而，若部分填充的扇区出现在二进制文件开头（即二进制文件未按4kB页面对齐），或部分填充的扇区位于二进制文件中间（即二进制文件稀疏或不连续），则该UF2文件可能会被错误写入。</p> <p>请注意，SDK所生成的绝大多数UF2文件确实按4kB边界对齐且连续，然而通过修改链接脚本或对静态数据施加极端对齐要求，SDK亦可能产生非对齐或非连续的二进制文件。其他语言或工具生成的二进制文件也可能不满足4kB对齐或连续存放的要求。</p>
解决方案	<p>解决方案是在包含某些页面数据的任何4kB扇区（除最后一个外）中包含所有页面的数据。</p> <p>从SDK版本1.3.1起，elf2uf2工具会自动处理此问题，明确向适当的部分填充扇区添加填充零页。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档 / 软件

时钟

RP2040-E7

参考	RP2040-E7
摘要	ROSC和XOSC COUNT寄存器不可靠
描述	ROSC和XOSC COUNT寄存器设计用于配置PHY及PLL等组件，适用于需要微秒级延迟且clk_sys频率可变时，NOP循环不足以满足的情况。但由于同步问题，ROSC:COUNT和XOSC:COUNT寄存器不可靠。
解决方案	请勿使用ROSC:COUNT或XOSC:COUNT寄存器
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复，禁止使用。C SDK未使用这些寄存器。

RP2040-E10

参考	RP2040-E10
----	------------

摘要	ROSC STATUS寄存器中的BADWRITE字段不可靠
描述	ROSCSTATUS寄存器中的BADWRITE字段意在报告向其他ROSC寄存器写入无效值的情况。然而，由于内部缺陷，ROSC:STATUS.BADWRITE字段不可靠。
解决方案	请勿使用ROSC:STATUS.BADWRITE字段。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复，禁止使用。C SDK不使用该字段。

DMA

RP2040-E12

参考	RP2040-E12
摘要	在地址环绕或非递增传输序列进行时，读取DMA的WRITE_ADDR和READ_ADDR寄存器会返回错误值。
描述	<p>DMA的内部WRITE_ADDR和READ_ADDR寄存器会在每次DMA向其总线流水线发出新地址时递增。若处理器在传输序列进行中读取这些寄存器，DMA返回的值将被调整，按进行中的传输数量（即已发送至总线流水线但尚未完成的传输）乘以单次传输字节数向下修正。</p> <p>该逻辑旨在确保读取READ_ADDR和WRITE_ADDR反映读/写操作已完成的位置，而非仅仅是地址已发出的位置。该逻辑未考虑到某些传输模式——尤其是当CTRL.INCR_WRITE == 0、CTRL.INCR_READ == 0或CTRL.RING_SIZE != 0时，READ_ADDR和WRITE_ADDR不会线性递增。</p>
解决方案	<p>与其检查READ_ADDR或WRITE_ADDR来监控传输序列的进度，不如检查TRANS_COUNT。</p> <p>TRANS_COUNT具有类似的传输中调整逻辑，但不受此缺陷影响，因为它始终线性递减。可根据READ_ADDR和WRITE_ADDR的初始值及TRANS_COUNT计算它们的正确值。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档

RP2040-E13

参考	RP2040-E13
摘要	在中止通道后，ABORT状态会被过早清除，且可能触发中断。
描述	<p>DMA的ABORT寄存器用于取消正在进行的传输序列，例如当通道卡在非活动的外设DREQ请求时。如果在触发中止时通道存在任何进行中的传输（即传输的读取周期已完成但写入周期未完成），则ABORT位不会等待这些传输完成即被清除。</p> <p>由于ABORT位被过早清除，当进行中的传输完成时，DMA将其视为正常完成。此操作设置通道的中断状态标志，前提是未设置CTRL.IRQ_QUIET。</p>

解决方案	在中止通道之前，应清除其中断使能。中止通道后，应轮询 CTRL.BUSY 位以等待操作完成（非 ABORT 位），清除虚假中断，并恢复中断使能。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

GPIO / ADC

RP2040-E6

参考	RP2040-E6
摘要	默认情况下，ADC引脚的GPIO数字输入未被禁用。
描述	GPIO26-29与ADC输入AIN0-3共用。RUN释放后，GPIO数字输入被启用。若将这些引脚接入模拟信号进行测量，可能会在这些引脚上产生意外的信号电平。通常这不会引起问题，因为数字输入默认启用迟滞效应。
解决方案	如使用模拟输入，应在启动后尽早禁用数字输入。此操作由RP2040B2的引导ROM以及RP2040B0和RP2040B1 SDK平台初始化代码的早期阶段完成。用户若需使用数字输入，必须手动启用。
影响	RP2040B0, RP2040B1
修复于	RP2040B2 启动只读存储器。该问题已在 SDK 中的 RP2040B0 和 RP2040B1 上修复。自定义用户代码应尽早禁用相关输入。

RP2040-E11

参考	RP2040-E11
摘要	ADC 中的 DNL 误差峰值
描述	RP2040 ADC 的 DNL 大致平坦且低于 1 LSB，但在四个数值——512、1536、2560 和 3584——处，其 DNL 误差峰值超过该值。ADC 的有效位数 (ENOB) 已由模拟的 9 位降低至实测的 8.7 位，详见第 4.9.3 节。DNL 误差将在一定程度上依据应用场景限制 ADC 性能。
解决方案	无
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复。

USB

RP2040-E2

参考	RP2040-E2
摘要	USB 设备端点中止状态未被清除。

描述	USB设备控制器（第4.1节）允许通过设置EP_ABORT寄存器中相应端点的位来中止该端点上任何待处理的事务。由于逻辑错误，如果对任何设置了EP_ABORT位的端点发起传输，USB设备控制器将不断对所有端点返回NAK。
解决方案	请勿使用EP_ABORT位。
影响	RP2040B0, RP2040B1
修复于	RP2040B2

RP2040-E3

参考	RP2040-E3
摘要	USB主机：中断端点的缓冲区完成标志可能在错误的缓冲区选择情况下被设置。
描述	USB主机有两种传输类型：正常的软件启动传输和中断传输，其中主机会在特定时间间隔后轮询中断端点。例如，每1毫秒轮询一次鼠标以检测其移动。中断传输为单缓冲模式，但控制器不会将缓冲区选择器重置为零。这意味着，如果发生软件启动传输，中断传输可能在选择了BUF1而非BUF0时触发缓冲完成标志。解决方案是忽略中断端点的BUFF_CPU_SHOULD_HANDLE寄存器。
解决方案	
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

RP2040-E4

参考	RP2040-E4
摘要	USB主机在单缓冲模式下写入缓冲状态的高半部分。
描述	USB主机维护一个缓冲区选择器，用于在BUF0和BUF1之间切换。此项设置应仅在双缓冲模式下切换，但在单缓冲模式下亦被切换。对于持续多个数据包的传输（即低速模式下长度超过8字节，全速模式下长度超过64字节），当缓冲区选择错误地设置为BUF1时，缓冲区状态可写回状态寄存器的BUF1半区。请注意，这不会影响从缓冲控制寄存器读取新的缓冲区信息，因为控制器在单缓冲模式下读取该寄存器时会忽略缓冲区选择器。
解决方案	如果缓冲区选择器为BUF1，则应将端点控制寄存器右移16位。您可以使用BUFF_CPU_SHOULD_HANDLE查找缓冲区被标记完成时的缓冲区选择器值。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

RP2040-E5

参考	RP2040-E5
摘要	USB设备无法在繁忙的USB总线上退出RESET状态。

描述	<p>USB总线的 RESET 状态由主机向设备发送持续10毫秒的 SE0信号触发。USB设备控制器在总线复位后需要800μs的空闲时间（J态），然后才能进入 CONNECTED状态。如果没有这段空闲时间，USB设备将无法连接，且不会接收主机发来的任何数据包，从而无法完成枚举。</p> <p>设备复位发生于设备插入之后。尽管主机会在与刚复位的设备通信之前进行等待，但接入同一USB集线器的其他设备可能已在与主机通信。</p> <p>USB 2.0和USB 3.0集线器配备一个或多个事务转换器，以便在高速总线上支持低速和全速事务。具体情况取决于集线器设计，但一个事务转换器通常由多个端口共享。</p> <p>由于RP2040的USB设备为全速设备，当其连接到集线器时，其通信流量将通过事务转换器传输。这意味着如果在RP2040旁边连接了另一设备，RP2040可能会接收到主机发给该设备的一些消息。如果设备活动不频繁，例如每8毫秒轮询一次的鼠标，则不会出现问题。然而某些设备，如USB串口，每30至50μs轮询一次。在这种情况下，总线活动非常频繁，会导致RP2040无法退出 RESET 状态并无法连接。</p> <p>RP2040B2中有硬件修正，避免了在 RESET 状态后需800μs空闲时间的情况。</p> <p>对此问题也有软件解决方案（详见解决方案部分）。用户还可通过在连接RP2040时关闭USB串口或其他有问题的设备，随后重新打开USB串口以规避该问题。</p> <p>在更大型的集线器中，将RP2040移至远离问题设备（即不同事务转换器）处，也可能解决该问题。例如，将RP2040连接至7端口集线器的端口1，将USB串口控制台连接至端口7，可能解决该问题。将RP2040连接至独立USB集线器上的任何繁忙设备，亦可解决该问题。</p>
解决方案	<p>通过软件强制USB设备控制器检测空闲USB总线800μs，将设备从 RESET 状态切换至 CONNECTED 状态。该修复利用连接至GPIO15的内部调试逻辑，作用时间为800μs。此举强制控制器将DP视为逻辑1（DM为逻辑0），使USB设备控制器判定USB总线上存在 J-state。此修复无须对GPIO15进行特定连接。我们亦可使用第2.19节中的输入覆盖功能，通过软件强制输入路径。详见 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_fix/rp2040_usb_deviceEnumeration/rp2040_usb_device_enumeration.c。</p> <p>注意该方案在设备重置期间会控制GPIO 15，使用前请确保设备重置阶段未将GPIO 15用于其他用途。设备在首次连接后会发生复位，但主机控制下的其他时间也可能发生复位。</p> <p>使用TinyUSB和SDK的解决方案十分简便，因为上述源文件已被 <code>pico_fix_rp2040_usb_device_enumeration</code> 库包含（该库作为设备模式下TinyUSB的自动依赖项添加）。该修复默认未启用，因修复方案使用GPIO 15，可能与应用程序自身对GPIO 15的使用产生冲突。您可以通过设置以下任一选项来启用该修复</p> <p>在您的 <code>CMakeLists.txt</code> 中将 <code>PICO_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> 加入编译器定义，或 在您的 <code>tusb_config.h</code> 中将 <code>TUD_OPT_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> 设置。</p> <p>即便使用内含硬件修复的RP2040版本，启用软件解决方案仍然安全且代价低廉。</p>
影响	RP2040B0, RP2040B1

修复于	RP2040B2。RP2040B0和RP2040B1上的软件解决方案。该解决方案未包含于bootrom中的USB大容量存储代码。该软件变通方案要求在USB总线复位期间使用GPIO15。
-----	--

RP2040-E15

参考	RP2040-E15
摘要	若在IN传输过程中发生特定的总线错误，USB设备控制器将会挂起。
描述	<p>若出现以下关键事件序列，USB设备控制器将进入不可恢复状态：</p> <ul style="list-style-type: none"> • RP2040连接至VL805 xHCI控制器，且以全速模式运行。 • 集成集线器检测到下游端口事务转换器流量与上游广播流量（帧起始令牌）之间即将发生线路碰撞。 • 集成集线器在下游进行中的数据包或令牌的PID或CRC部分故意制造比特填充错误。 <p>此序列已知发生于Raspberry Pi 4或Raspberry Pi 400的下游端口，以及数据缓冲区大小超过50字节的Bulk IN端点。在此情况下，集成USB 2.0集线器错误判断主机SOF包的剩余全速帧时间，误发IN令牌，导致后续ACK响应被破坏并被传播的SOF包替代。</p> <p>此类数据损坏未被设备状态机正确处理，设备控制器必须重置。</p> <p>此序列未见发生在普通USB 2.0集线器上，也未见于非VL805 xHCI控制器提供的根端口。</p>
解决方案	<p>1) VL805固件版本 0138c1</p> <p>已向Pi 4系列产品的raspberrypi-bootloaderApt软件包中的DEFAULT频道推送了更新固件。该固件修正了错误的集线器时间计算。此固件更新不会自动应用，用户须在Pi 4设备上运行sudo rpi-eeprom-update -a命令并按照屏幕提示操作。</p> <p>2) Linux内核 xHCI 驱动补丁</p> <p>针对VL805固件版本早于0138c1的Pi 4系列产品，发布了内核更新，避免在全速帧最后微帧期间向受影响端点的控制器排入单一传输描述符。此更新包含于raspberrypi-kernelApt软件包中。</p> <p>2) SDK v1.5.0 / TinyUSB 0.15.0</p> <p>从版本0.15.0开始，TinyUSB引入了解决该错误的规避方案，此版本已纳入SDK v1.5.0发布版。dcd_rp2040驱动程序将在全速帧的最后200μs内避免启用Bulk IN缓冲区。此做法使可用的Bulk IN带宽约减少20%，并有选择性地启用帧起始中断。</p> <p>对于不会连接到受影响VL805端口的实现，例如RP2040直接连接至板载集线器的电路设计，Tiny USB的规避方案并非必需。可通过在tusb_config.h中定义TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX=0来禁用该规避方案。</p>

影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

RP2040-E16

参考	RP2040-E16
摘要	USB状态信号同步不足
描述	<p>在USB外设中，某些主控制器与设备控制器事件会从 <code>clk_usb</code> 跨越至 <code>clk_sys</code>。许多此类信号缺乏适当的同步方法，无法确保当 <code>clk_sys</code> 等于或低于 <code>clk_usb</code> 时信号被正确采集。</p> <p>以下信号缺少适当的同步机制：</p> <p><code>SIE_STATUS:</code></p> <p><code>* TRANS_COMPLETE * SETUP_REC * STALL_REC * NAK_REC * RX_SHORT_PACKET * ACK_REQ * DATA_SEQ_ERROR * RX_OVERFLOW</code></p> <p><code>INTR:</code></p> <p><code>* HOST_SOF * ERROR_CRC * ERROR_BIT_STUFF * ERROR_RX_OVERFLOW * ERROR_RX_TIMEOUT * ERROR_DATA_SEQ</code></p> <p>Bootrom的USB引导加载程序将 <code>clk_sys</code> 从 <code>clk_usb</code> 链式传递，因此两者时钟频率一致且具有固定相位关系。在该条件及PVT极限情况下，实验室测试发现这些事件可能丢失，导致USB引导加载程序表现不稳定。</p>
解决方案	当外设使用时， <code>clk_sys</code> 的运行速度至少比 <code>clk_usb</code> 快10%。诸如复位、挂起和恢复等准静态总线状态的信号不受该缺陷影响，因此在这些情况下 <code>clk_sys</code> 可降低。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档，软件。该问题未在RP2040引导只读存储器（bootrom）中修复。

看门狗

RP2040-E1

参考	RP2040-E1
摘要	看门狗计数每个时钟周期减少两次。
描述	看门狗（第4.7节）具有一个24位计数器，从用户在LOAD寄存器中设定的值开始，每个时钟周期递减一次。存在逻辑错误，导致计数器每个时钟周期递减两次，而非一次。在推荐配置下，时钟周期间隔为1μs，该缺陷使看门狗计数器重置的最大间隔时间从约16.7秒减半至约8.3秒。
解决方案	在LOAD寄存器中设置预期值的两倍。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

XIP Flash

RP2040-E8

参考	RP2040-E8
摘要	中止XIP DMA流并立即启动新流时存在竞态条件。
描述	<p>XIP DMA 流式硬件允许在后台按线性顺序进行闪存读取，由 DMA 读取，且不会因普通 XIP 窗口访问而引起的总线阻塞影响 DMA。通过先写入 STREAM_ADDR 寄存器，再写入 STREAM_CTR，来启动一个数据流；可通过向 STREAM_CTR 写入 0 来中途停止该数据流。</p> <p>当以此方式中止数据流时，软件有足够时间加载新地址并启动新数据流，而中止数据流的最后一次 SPI/QSPI 访问仍在进行中。这会导致新加载的数据流地址在新数据流序列的第一次数据传输之前递增一次，因此整个数据流以 4 字节偏移进行。</p>
解决方案	清除 STREAM_CTR 后，立即从未缓存的 XIP 窗口执行一次虚拟读取，例如 <code>(void)*(io_ro_32*)XIP_NOCACHE_NOALLOC_BASE;</code> 。如果 XIP 流传输仍在进行中，该虚假读取将阻塞，直到传输完成。然后，先写入 STREAM_ADDR，随后写入 STREAM_CTR，即可安全开始新的流。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

附录 C：供应情况

Raspberry Pi 理解客户对产品长期可用性的重视，故致力于在实际可行范围内持续供应。我们预计 RP2040 最少会持续生产至 2041 年 1 月。

支持

有关支持，请参阅 Raspberry Pi 网站的 Pico 部分，并在 Raspberry Pi 论坛发布相关问题。

订购代码

RP2040 可通过 Raspberry Pi Direct 批量订购。

表 638. 零件
编号

型号	订购代码	最低订购 数量	建议零售价	等效每芯片 价格
7" 卷带，含 500 颗 RP2040 芯片	SC0914(7)	1 件以上 / 批量	400.00 美元	0.80 美元
13" 卷带，含 3400 颗 RP2040 芯片	SC0914(13)	1 件以上 / 批量	US\$2,380.00	US\$0.70

① 注意

建议零售价在出版时为正确且未含税款。

文档发行历史

2025年2月20日

- 更新了寄存器字段类型描述，采用改进后的RP2350措辞。
- 新增有关双核Cortex M0+处理器核心以200MHz频率运行的信息。

2024年10月15日

- 修正了轻微拼写错误及格式问题。
- 恢复为每个PDF独立的发布历史记录。

2024年5月2日

- 修正了轻微拼写错误及格式问题。

2024年2月2日

- 修正了轻微拼写错误及格式问题。
- 更新了ROSC寄存器信息。
- 更新内容包括用于RP2040晶体的新推荐零件号。

2023年6月14日

- 修正了轻微拼写错误及格式问题。

2023年3月3日

- 修正了轻微拼写错误及格式问题。
- 新增勘误表E15。
- 新增封装标记规格。
- 新增RP2040基线功耗数据。

2022年12月1日

- 修正了轻微拼写错误及格式问题。
- 新增RP2040供应情况说明。
- 新增RP2040存储条件及热性能资料。

- 将SDK库文档替换为在线版本链接。

2022年6月30日

- 修正了轻微拼写错误及格式问题。

2022年6月17日

- 修正了轻微拼写错误及格式问题。
- RP2040现获-40°C认证，最低工作温度由-20°C调整至-40°C。
- 为提升工作条件下的稳定性，将PLL最小VCO频率自400MHz提高至750MHz。
- 新增勘误 E12、E13 和 E14。

2021年11月4日

- 修正了轻微拼写错误及格式问题。
- 改进了有关 USB 双缓冲的文档内容。
- 更新了文档中的链接。

2021年11月3日

- 修正了轻微拼写错误及格式问题。
- 修正了部分寄存器访问类型及其描述。
- 添加了核心 1 启动序列的相关信息。
- 阐述了 SDK 的 "panic" 处理机制。
- 更新了 `picotool` 工具的文档。

2021年9月30日

- 修正了轻微拼写错误及格式问题。
- 新增了关于 B2 版本发布的说明。
- 更新了 B2 版本的勘误信息。

2021年6月23日

- 修正了轻微拼写错误及格式问题。
- 更新了 ADC 相关信息。
- 新增勘误 E11。

2021年6月7日

- 修正了轻微拼写错误及格式问题。
- 添加了SDK的发布历史记录。

2021年4月13日

- 修正了轻微拼写错误及格式问题。
- 明确规定文档中所有源代码均遵循3-Clause BSD许可证。

2021年4月7日

- 修正了轻微拼写错误及格式问题。
- 新增勘误E10。

2021年3月5日

- 修正了轻微拼写错误及格式问题。
- 优化了引脚排列图。

2021年2月23日

- 修正了轻微拼写错误及格式问题。
- 更换了字体。
- 新增关于RP2040输入/输出极限的附加文档。
- 对SWD文档内容进行了重大改进。
- 新增勘误表E7、E8及E9。

2021年2月1日

- 修正了轻微拼写错误及格式问题。
- 对PIO文档进行小幅改进。
- 在DMA中补充了缺失的TIMER2及TIMER3寄存器。

2021年1月26日

- 修正了轻微拼写错误及格式问题。
- 新增关于在ADC中使用DMA的详细信息。
- 澄清了M0+与SIO CPUID寄存器。
- 增加了关于定时器的详细讨论。

- 重命名书籍并优化了输出PDF文件的大小。

2021年1月21日

- 初始发布。



Raspberry Pi is a trademark of Raspberry Pi Ltd

Raspberry Pi Ltd